

The `ltemplates.dtx` code*

Frank Mittelbach, Chris Rowley, David Carlisle, L^AT_EX Project[†]

October 31, 2025

1 Introduction

There are three broad “layers” between putting down ideas into a source file and ending up with a typeset document. These layers of document writing are

1. authoring of the text with mark-up;
2. document layout design;
3. implementation (with T_EX programming) of the design.

We write the text as an author, and we see the visual output of the design after the document is generated; the T_EX implementation in the middle is the glue between the two.

L^AT_EX’s greatest success has been to standardise a system of mark-up that balances the trade-off between ease of reading and ease of writing to suit almost all forms of technical writing. It’s other original strength was a good background in typographical design; while the standard L^AT_EX 2_ε classes look somewhat dated now in terms of their visual design, their typography is generally sound (barring the occasional minor faults).

However, L^AT_EX 2_ε has always lacked a standard approach to customising the visual design of a document. Changing the looks of the standard classes involved either:

- Creating a new version of the implementation code of the class and editing it.
- Loading one of the many packages to customise certain elements of the standard classes.
- Loading a completely different document class, such as KOMA-Script or memoir, that allows easy customization.

All three of these approaches have their drawbacks and learning curves.

The idea behind `ltemplates` is to cleanly separate the three layers introduced at the beginning of this section, so that document authors who are not programmers can easily change the design of their documents. `ltemplates` also makes it easier for L^AT_EX programmers to provide their own customizations on top of a pre-existing class.

*This file has version v1.0f dated 2025-07-08, © L^AT_EX Project.

†E-mail: latex-team@latex-project.org

2 What is a document?

Besides the textual content of the words themselves, the source file of a document contains mark-up elements that add structure to the document. These elements include sectional divisions, figure/table captions, lists of various sorts, theorems/proofs, and so on. The list will be different for every document that can be written.

Each element can be represented logically without worrying about the formatting, with mark-up such as `\section`, `\caption`, `\begin{enumerate}` and so on. The output of each one of these document elements will be a typeset representation of the information marked up, and the visual arrangement and design of these elements can vary widely in producing a variety of desired outcomes.

For each type of document element, there may be design variations that contain the same sort of information but present it in slightly different ways. For example, the difference between a numbered and an unnumbered section, `\section` and `\section*`, or the difference between an itemized list or an enumerated list.

There are three distinct layers in the definition of “a document” at this level

1. semantic elements such as the ideas of sections and lists;
2. a set of design solutions for representing these elements visually;
3. specific variations for these designs that represent the elements in the document.

In the parlance of the template system, these are called types, templates, and instances, and they are discussed below in sections 4, 5, and 7, respectively.

3 Types, templates, and instances

By formally declaring documents to be composed of mark-up elements grouped into types, which are interpreted and typeset with a set of templates, each of which has one or more instances with which to compose each and every semantic unit of the text, we can cleanly separate the components of document construction.

All of the structures provided by the template system are global, and do not respect \TeX grouping.

4 Template types

An *template type* (sometimes just “type”) is an abstract idea of a document element that takes a fixed number of arguments corresponding to the information from the document author that it is representing. A sectioning type, for example, might take three inputs: “title”, “short title”, and “label”.

Any given document class will define which types are to be used in the document, and any template of a given type can be used to generate an instance for the type. (Of course, different templates will produce different typeset representations, but the underlying content will be the same.)

`\NewTemplateType` `\NewTemplateType {template type} {no. of args}`

This function defines an *template type* taking *number of arguments*, where the *type* is an abstraction as discussed above. For example,

```
\NewTemplateType{sectioning}{3}
```

creates a type “sectioning”, where each use of that type will need three arguments.

5 Templates

A *template* is a generalized design solution for representing the information of a specified type. Templates that do the same thing, but in different ways, are grouped together by their type and given separate names. There are two important parts to a template:

- the parameters it takes to vary the design it is producing;
- the implementation of the design.

As a document author or designer does not care about the implementation but rather only the interface to the template, these two aspects of the template definition are split into two independent declarations, `\DeclareTemplateInterface` and `\DeclareTemplateCode`.

`\DeclareTemplateInterface` `\DeclareTemplateInterface`
`{type} {template} {no. of args}`
`{key list}`

A *template* interface is declared for a particular *type*, where the *number of arguments* must agree with the type declaration. The interface itself is defined by the *key list*, which is itself a key–value list taking a specialized format:

```
key1 : key type1 ,  
key2 : key type2 ,  
key3 : key type3 = default3 ,  
key4 : key type4 = default4 ,  
...
```

Each *key* name should consist of ASCII characters, with the exception of `,` `=` and `:`. The recommended form for key names is to use lower case letters, with dashes to separate out different parts. Spaces are ignored in key names, so they can be included or missed out at will. Each *key* must have a *key type*, which defines the type of input that the *key* requires. A full list of key types is given in Table 1. Each key may have a *default* value, which will be used in by the template if the *key* is not set explicitly. The *default* should be of the correct form to be accepted by the *key type* of the *key*: this is not checked by the code. Expressions for numerical values are evaluated when the template is used, thus for example values given in terms of `em` or `ex` will be set respecting the prevailing font.

Key-type	Description of input
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>choice{⟨choices⟩}</code>	A list of pre-defined <code>⟨choices⟩</code>
<code>commalist</code>	A comma-separated list
<code>function{⟨N⟩}</code>	A function definition with N arguments (N from 0 to 9)
<code>instance{⟨name⟩}</code>	An instance of type <code>⟨name⟩</code>
<code>integer</code>	An integer or integer expression
<code>length</code>	A fixed length
<code>muskip</code>	A math length with shrink and stretch components
<code>real</code>	A real (floating point) value
<code>skip</code>	A length with shrink and stretch components
<code>tokenlist</code>	A token list: any text or commands

Table 1: Key-types for defining template interfaces with `\DeclareTemplateInterface`.

`\KeyValue` `\KeyValue {⟨key name⟩}`

There are occasions where the default (or value) for one key should be taken from another. The `\KeyValue` function can be used to transfer this information without needing to know the internal implementation of the key:

```

\DeclareTemplateInterface { type } { template } { no. of args }
{
  key-name-1 : key-type = value ,
  key-name-2 : key-type = \KeyValue { key-name-1 },
  ...
}

```

Key-type	Description of binding
<code>boolean</code>	Boolean variable, <i>e.g.</i> <code>\l_tmpa_bool</code>
<code>choice</code>	List of choice implementations (see Section 6)
<code>commalist</code>	Comma list, <i>e.g.</i> <code>\l_tmpa_clist</code>
<code>function</code>	Function taking N arguments, <i>e.g.</i> <code>\use_i:nn</code>
<code>instance</code>	
<code>integer</code>	Integer variable, <i>e.g.</i> <code>\l_tmpa_int</code>
<code>length</code>	Dimension variable, <i>e.g.</i> <code>\l_tmpa_dim</code>
<code>muskip</code>	Muskip variable, <i>e.g.</i> <code>\l_tmpa_muskip</code>
<code>real</code>	Floating-point variable, <i>e.g.</i> <code>\l_tmpa_fp</code>
<code>skip</code>	Skip variable, <i>e.g.</i> <code>\l_tmpa_skip</code>
<code>tokenlist</code>	Token list variable, <i>e.g.</i> <code>\l_tmpa_tl</code>

Table 2: Bindings required for different key types when defining template implementations with `\DeclareTemplateCode`. Apart from `code`, `choice` and `function` all of these accept the key word `global` to carry out a global assignment.

```
\DeclareTemplateCode \DeclareTemplateCode
  <{type}> <{template}> <{no. of args}>
  <{key bindings}> <{code}>
```

The relationship between a templates keys and the internal implementation is created using the `\DeclareTemplateCode` function. As with `\DeclareTemplateInterface`, the `<template>` name is given along with the `<type>` and `<number of arguments>` required. The `<key bindings>` argument is a key–value list which specifies the relationship between each `<key>` of the template interface with an underlying `<variable>`.

```
<key1> = <variable1>,
<key2> = <variable2>,
<key3> = global <variable3>,
<key4> = global <variable4>,
...
```

With the exception of the `choice`, `code` and `function` key types, the `<variable>` here should be the name of an existing L^AT_EX₃ register. As illustrated, the key word “global” may be included in the listing to indicate that the `<variable>` should be assigned globally. A full list of variable bindings is given in Table 2.

The `<code>` argument of `\DeclareTemplateCode` is used as the replacement text for the template when it is used, either directly or as an instance. This may therefore accept arguments `#1`, `#2`, *etc.* as detailed by the `<number of arguments>` taken by the type.

```
\AssignTemplateKeys \AssignTemplateKeys
```

In the final argument of `\DeclareTemplateCode` the assignment of keys defined by the template may be delayed by including the command `\AssignTemplateKeys`. If this is *not* present, keys are assigned immediately before the template code. If an `\AssignTemplateKeys` command is present, assignment is delayed until this point. Note that the command must be *directly* present in the code, not placed within a nested command/macro.

<code>\SetKnownTemplateKeys</code>	<code>\SetKnownTemplateKeys {<type>} {<template>} {<keyvals>}</code>
<code>\SetTemplateKeys</code>	<code>\SetTemplateKeys {<type>} {<template>} {<keyvals>}</code>
<code>\UnusedTemplateKeys</code>	<code>\UnusedTemplateKeys % all <keyvals> unused by previous \SetKnownTemplateKeys</code>

In the final argument of `\DeclareTemplateCode` one can also overwrite (some of) the current template key value settings by using the command `\SetKnownTemplateKeys` or `\SetTemplateKeys`, i.e., they can overwrite the template default values and the values assigned by the instance.

The `\SetKnownTemplateKeys` and `\SetTemplateKeys` commands are only supported within the code of a template; using them elsewhere has unpredictable results. If they are used together with `\AssignTemplateKeys` then the latter command should come first in the template code.

The main use case for these commands is the situation where there is an argument (normally #1) to the template in which a key/value list can be specified that overwrites the normal settings. In that case one could use

```
\SetKnownTemplateKeys{<type>}{<template>}{#1}
```

to process this key/value list inside the template.

If `\SetKnownTemplateKeys` is executed and the `<keyvals>` argument contains keys not known to the `<template>` they are simply ignored and stored in the tokenlist `\UnusedTemplateKeys` without generating an error. This way it is possible to apply the same key/val list specified by the user on a document-level command or environment to several templates, which is useful, if the command or environment is implemented by calling several different template instances.

As a variation of that, you can use this key/val list the first time, and for the next template instance use what remains in `\UnusedTemplateKeys` (i.e., the key/val list with only the keys that have not been processed previously). The final processing step could then be `\SetTemplateKeys`, which unconditionally attempts to set the `<keyvals>` received in its third argument. This command complains if any of them are unknown keys. Alternatively, you could use `\SetKnownTemplateKeys` and afterwards check whether `\UnusedTemplateKeys` is empty.¹

For example, a list, such as `enumerate`, is made up from a `blockenv`, `block`, `list`, and a `para` template and in the single user-supplied optional argument of `enumerate` key/values for any of these templates might be specified.

In fact, in the particular example of list environments, the supplied key/value list is also saved and then applied to each `\item` which is implemented through an `item` template. This way, one can specify one-off settings for all the items of a single list (on the environment level), as well as to individual items within that list (by specifying them in the optional argument of an `\item`). With `\SetKnownTemplateKeys` and `\SetTemplateKeys` working together, it is possible to provide this flexibility and still alert the user when one of their keys is misspelled.

On the other hand you may want to allow for “misspellings” without generating an error or a warning. For example, if you define a template that accepts only a few keys, you might just want to ignore anything specified in the source when you use this template in place of a different one, without the need to alter the document source. Or you might

¹Using `\SetTemplateKeys` exposes the inner structure of the template keys when generating an error. This is something one may want to avoid as it can be confusing to the user, especially if several templates are involved. In that case use `\SetKnownTemplateKeys` and afterwards check whether `\UnusedTemplateKeys` is empty; if it is not empty then generate your own error message.

just generate a warning message, which is easy, given that the unused key/values are available in the `\UnusedTemplateKeys` variable.

```
\DeclareTemplateCopy \DeclareTemplateCopy
  {<type>} {<template2>} {<template1>}
```

Copies `<template1>` of `<type>` to a new name `<template2>`: the copy can then be edited independent of the original.

6 Multiple choices

The `choice` key type implements multiple choice input. At the interface level, only the list of valid choices is needed:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
  { key-name : choice { A, B, C } }
```

where the choices are given as a comma-list (which must therefore be wrapped in braces). A default value can also be given:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
  { key-name : choice { A, B, C } = A }
```

At the implementation level, each choice is associated with code, using a nested key-value list.

```
\DeclareTemplateCode { foo } { bar } { 0 }
  {
    key-name =
    {
      A = Code-A ,
      B = Code-B ,
      C = Code-C
    }
  }
  { ... }
```

The two choice lists should match, but in the implementation a special `unknown` choice is also available. This can be used to ignore values and implement an “else” branch:

```
\DeclareTemplateCode { foo } { bar } { 0 }
  {
    key-name =
    {
      A      = Code-A ,
      B      = Code-B ,
      C      = Code-C ,
      unknown = Else-code
    }
  }
  { ... }
```

The `unknown` entry must be the last one given, and should *not* be listed in the interface part of the template.

For keys which accept the values `true` and `false` both the boolean and choice key types can be used. As template interfaces are intended to prompt clarity at the design level, the boolean key type should be favored, with the choice type reserved for keys which take arbitrary values.

7 Instances

After a template is defined it still needs to be put to use. The parameters that it expects need to be defined before it can be used in a document. Every time a template has parameters given to it, an *instance* is created, and this is the code that ends up in the document to perform the typesetting of whatever pieces of information are input into it.

For example, a template might say “here is a section with or without a number that might be centered or left aligned and print its contents in a certain font of a certain size, with a bit of a gap before and after it” whereas an instance declares “this is a section with a number, which is centered and set in 12pt italic with a 10pt skip before and a 12pt skip after it”. Therefore, an instance is just a frozen version of a template with specific settings as chosen by the designer.

```
\DeclareInstance \DeclareInstance
  {<type>} {<instance>} {<template>} {<parameters>}
```

This function uses a `<template>` for an `<type>` to create an `<instance>`. The `<instance>` will be set up using the `<parameters>`, which will set some of the `<keys>` in the `<template>`.

As a practical example, consider a type for document sections (which might include chapters, parts, sections, *etc.*), which is called `sectioning`. One possible template for this type might be called `basic`, and one instance of this template would be a numbered section. The instance declaration might read:

```
\DeclareInstance { sectioning } { section-num } { basic }
{
  numbered      = true ,
  justification = center ,
  font          = \normalsize\itshape ,
  before-skip   = 10pt ,
  after-skip    = 12pt ,
}
```

Of course, the key names here are entirely imaginary, but illustrate the general idea of fixing some settings.

```
\IfInstanceExistsT \IfInstanceExistsTF {<type>} {<instance>} {<>true code>} {<>false code>}
\IfInstanceExistsF
\IfInstanceExistsTF
```

Tests if the named `<instance>` of a `<type>` exists, and then inserts the appropriate code into the input stream.

```
\DeclareInstanceCopy \DeclareInstanceCopy
  {<type>} {<instance2>} {<instance1>}
```

Copies the `<values>` for `<instance1>` for an `<type>` to `<instance2>`.

8 Document interface

After the instances have been chosen, document commands must be declared to use those instances in the document. `\UseInstance` calls instances directly, and this command should be used internally in document-level mark-up.

```
\UseInstance \UseInstance
              {<type>} {<instance>} <arguments>
```

Uses an `<instance>` of the `<type>`, which will require `<arguments>` as determined by the number specified for the `<type>`. The `<instance>` must have been declared before it can be used, otherwise an error is raised.

```
\UseTemplate \UseTemplate {<type>} {<template>}
              {<settings>} <arguments>
```

Uses the `<template>` of the specified `<type>`, applying the `<settings>` and absorbing `<arguments>` as detailed by the `<type>` declaration. This in effect is the same as creating an instance using `\DeclareInstance` and immediately using it with `\UseInstance`, but without the instance having any further existence. This command is therefore useful when a template needs to be used only once.

This function can also be used as the argument to `instance` key types:

```
\DeclareInstance { type } { template } { instance }
{
  instance-key =
    \UseTemplate { type2 } { template2 } { <settings> }
}
```

9 Changing existing definitions

Template parameters may be assigned specific defaults for instances to use if the instance declaration doesn't explicit set those parameters. In some cases, the document designer will wish to edit these defaults to allow them to "cascade" to the instances. The alternative would be to set each parameter identically for each instance declaration, a tedious and error-prone process.

```
\EditTemplateDefaults \EditTemplateDefaults
                      {<type>} {<template>} {<new defaults>}
```

Edits the `<defaults>` for a `<template>` for an `<type>`. The `<new defaults>`, given as a key-value list, replace the existing defaults for the `<template>`. This means that the change will apply to instances declared after the editing, but that instances which have already been created are unaffected.

```
\EditInstance \EditInstance
              {<type>} {<instance>} {<new values>}
```

Edits the `<values>` for an `<instance>` for an `<type>`. The `<new values>`, given as a key-value list, replace the existing values for the `<instance>`. This function is complementary to `\EditTemplateDefaults`: `\EditInstance` changes a single instance while leaving the template untouched.

9.1 Expanding the values of keys

To allow the user to apply expansion of values when the key is set, key names can be followed by an expansion specifier. This is given by appending `:` and a single letter specifier to the key name. These letters are the normal argument specifiers for `expl3`, thus they may be one of `n` (redundant but supported), `o`, `V`, `v`, `e`, `N` (again redundant) or `c`. Expansion of a control sequence name is particularly useful when you need to refer to an internal \LaTeX 2_{ϵ} or an L3 programming layer variable, e.g.,

```
key-a:c = @itemdepth , % use \@itemdepth as the value
key-b:v = @itemdepth   % use the current value of \@itemdepth as the value
```

10 Getting information about templates and instances

<hr/> <hr/>	<code>\ShowInstanceValues</code>	<code>\ShowInstanceValues {<type>} {<instance>}</code>	Shows the <i><values></i> for an <i><instance></i> of the given <i><type></i> at the terminal.
<hr/> <hr/>	<code>\ShowTemplateCode</code>	<code>\ShowTemplateCode {<type>} {<template>}</code>	Shows the <i><code></i> of a <i><template></i> for an <i><type></i> in the terminal.
<hr/> <hr/>	<code>\ShowTemplateDefaults</code>	<code>\ShowTemplateDefaults {<type>} {<template>}</code>	Shows the <i><default></i> values of a <i><template></i> for an <i><type></i> in the terminal.
<hr/> <hr/>	<code>\ShowTemplateInterface</code>	<code>\ShowTemplateInterface {<type>} {<template>}</code>	Shows the <i><keys></i> and associated <i><key types></i> of a <i><template></i> for an <i><type></i> in the terminal.
<hr/> <hr/>	<code>\ShowTemplateVariables</code>	<code>\ShowTemplateVariables {<type>} {<template>}</code>	Shows the <i><variables></i> and associated <i><keys></i> of a <i><template></i> for an <i><type></i> in the terminal. Note that <code>code</code> and <code>choice</code> keys do not map directly to variables but to arbitrary code. For <code>choice</code> keys, each valid choice is shown as a separate entry in the list, with the key name and choice separated by a space, for example Template 'example' of type 'example' has variable mapping: > demo unknown => \def \demo {?} > demo c => \def \demo {c} > demo b => \def \demo {b} > demo a => \def \demo {a}.

would be shown for a choice key `demo` with valid choices `a`, `b` and `c`, plus code for an `unknown` branch.

11 The implementation

```
1 <@@=template>
2 <*2ekernel>
3 \message{templates,}
4 </2ekernel>
5 <*2ekernel | latexrelease>
6 \ExplSyntaxOn
7 <latexrelease>\NewModuleRelease{2024/06/01}{lttemplates}
8 <latexrelease> {Prototype-document~commands}%
```

11.1 Variables and constants

```
\c__template_code_root_tl
\c__template_defaults_root_tl
\c__template_instances_root_tl
\c__template_keytypes_root_tl
\c__template_key_order_root_tl
\c__template_restrict_root_tl
\c__template_values_root_tl
\c__template_vars_root_tl
```

So that literal values are kept to a minimum.

```
9 \tl_const:Nn \c__template_code_root_tl { template~code~>~ }
10 \tl_const:Nn \c__template_defaults_root_tl { template~defaults~>~ }
11 \tl_const:Nn \c__template_instances_root_tl { template~instance~>~ }
12 \tl_const:Nn \c__template_keytypes_root_tl { template~key~types~>~ }
13 \tl_const:Nn \c__template_key_order_root_tl { template~key~order~>~ }
14 \tl_const:Nn \c__template_values_root_tl { template~values~>~ }
15 \tl_const:Nn \c__template_vars_root_tl { template~vars~>~ }
```

```
\c__template_keytypes_arg_seq
```

A list of keytypes which also need additional data (an argument), used to parse the keytype correctly.

```
16 \seq_const_from_clist:Nn \c__template_keytypes_arg_seq
17 { choice , function , instance }
```

```
\g__template_type_prop
```

 For storing types and the associated number of arguments.

```
18 \prop_new:N \g__template_type_prop
```

```
\l__template_assignments_tl
```

When creating an instance, the assigned values are collected here.

```
19 \tl_new:N \l__template_assignments_tl
```

```
\l__template_default_tl
```

 The default value for a key is recovered here from the property list in which it is stored.

```
20 \tl_new:N \l__template_default_tl
```

\l__template_error_bool A flag for errors to be carried forward.

21 \bool_new:N \l__template_error_bool

\l__template_global_bool Used to indicate that assignments should be global.

22 \bool_new:N \l__template_global_bool

\l__template_key_name_tl
\l__template_keytype_tl
\l__template_keytype_arg_tl
\l__template_value_tl
\l__template_var_tl

When defining each key in a template, the name and type of the key need to be separated and stored. Any argument needed by the keytype is also stored separately.

23 \tl_new:N \l__template_key_name_tl
24 \tl_new:N \l__template_keytype_tl
25 \tl_new:N \l__template_keytype_arg_tl
26 \tl_new:N \l__template_value_tl
27 \tl_new:N \l__template_var_tl

\l__template_value_exp_str

28 \str_new:N \l__template_value_exp_str

\l__template_keytypes_prop To avoid needing too many difficult-to-follow csname assignments, various scratch token
\l__template_key_order_seq registers are used to build up data, which is then transferred

\l__template_values_prop
\l__template_vars_prop

29 \prop_new:N \l__template_keytypes_prop
30 \seq_new:N \l__template_key_order_seq
31 \prop_new:N \l__template_values_prop
32 \prop_new:N \l__template_vars_prop

\l__template_tmp_clist Scratch space.

\l__template_tmp_dim
\l__template_tmp_int
\l__template_tmp_muskip
\l__template_tmp_skip
\l__template_tmp_tl

33 \clist_new:N \l__template_tmp_clist
34 \dim_new:N \l__template_tmp_dim
35 \int_new:N \l__template_tmp_int
36 \muskip_new:N \l__template_tmp_muskip
37 \skip_new:N \l__template_tmp_skip
38 \tl_new:N \l__template_tmp_tl

\s__template_mark Internal scan marks.

\s__template_stop

39 \scan_new:N \s__template_mark
40 \scan_new:N \s__template_stop

`\q__template_nil` Internal quarks.

```
41 \quark_new:N \q__template_nil
```

`__template_quark_if_nil_p:n` Branching quark conditional.

```
42 \__kernel_quark_new_conditional:Nn \__template_quark_if_nil:N { F }
```

(End of definition for `__template_quark_if_nil:nTF`.)

11.2 Testing existence and validity

There are a number of checks needed for either the existence of a type, template or instance. There are also some for the validity of a particular call. All of these are collected up here.

`__template_execute_if_arg_agree:nnT` A test agreement between the number of arguments for the template type and that specified when creating a template. This is not done as a separate conditional for efficiency and better error message

```
43 \cs_new_protected:Npn \__template_execute_if_arg_agree:nnT #1#2#3
44 {
45   \prop_get:NnN \g__template_type_prop {#1} \l__template_tmp_tl
46   \int_compare:nNnTF {#2} = \l__template_tmp_tl
47     {#3}
48     {
49       \msg_error:nnee { template } { argument-number-mismatch }
50       {#1} { \l__template_tmp_tl } {#2}
51     }
52 }
```

(End of definition for `__template_execute_if_arg_agree:nnT`.)

`__template_execute_if_code_exist:nnT` A template is only fully declared if the code has been set up, which can be checked by looking for the template function itself.

```
53 \cs_new_protected:Npn \__template_execute_if_code_exist:nnT #1#2#3
54 {
55   \cs_if_exist:cTF { \c__template_code_root_tl #1 / #2 }
56     {#3}
57     { \msg_error:nenn { template } { no-template-code } {#1} {#2} }
58 }
```

(End of definition for `__template_execute_if_code_exist:nnT`.)

`__template_execute_if_keytype_exist:nT`
`__template_execute_if_keytype_exist:VT` The test for valid keytypes looks for a function to set up the key, which is part of the “code” side of the template definition. This avoids having different lists for the two parts of the process.

```
59 \cs_new_protected:Npn \__template_execute_if_keytype_exist:nT #1#2
60 {
61   \cs_if_exist:cTF { __template_store_value_ #1 :n }
62     {#2}
63     { \msg_error:nnn { template } { unknown-keytype } {#1} }
64 }
65 \cs_generate_variant:Nn \__template_execute_if_keytype_exist:nT { V }
```

(End of definition for `__template_execute_if_keytype_exist:nT`.)

`__template_execute_if_type_exist:nT` To check that a particular type is valid.

```
66 \cs_new_protected:Npn \__template_execute_if_type_exist:nT #1#2
67 {
68   \prop_if_in:NnTF \g__template_type_prop {#1}
69   {#2}
70   { \msg_error:nnn { template } { unknown-type } {#1} }
71 }
```

(End of definition for `__template_execute_if_type_exist:nT`.)

`__template_execute_if_keys_exist:nnT` To check that the keys for a template have been set up before trying to create any code, a simple check for the correctly-named keytype property list.

```
72 \cs_new_protected:Npn \__template_if_keys_exist:nnT #1#2#3
73 {
74   \cs_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
75   {#3}
76   { \msg_error:nnnn { template } { unknown-template } {#1} {#2} }
77 }
```

(End of definition for `__template_execute_if_keys_exist:nnT`.)

`__template_if_key_value:nTF` Tests for the first token in a string being `\KeyValue`.

`__template_if_key_value:VTF`

```
78 \prg_new_conditional:Npnn \__template_if_key_value:n #1 { T , F , TF }
79 {
80   \str_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_nil \q_stop }
81   \prg_return_true:
82   \prg_return_false:
83 }
84 \prg_generate_conditional_variant:Nnn \__template_if_key_value:n { V } { T , F , TF }
```

(End of definition for `__template_if_key_value:nTF`.)

`__template_if_instance_exist:nnTF` Testing for an instance

```
85 \prg_new_conditional:Npnn \__template_if_instance_exist:nn #1#2 { T, F, TF }
86 {
87   \cs_if_exist:cTF { \c__template_instances_root_tl #1 / #2 }
88   \prg_return_true:
89   \prg_return_false:
90 }
```

(End of definition for `__template_if_instance_exist:nnTF`.)

`__template_if_use_template:nTF` Tests for the first token in a string being `\UseTemplate`.

```
91 \prg_new_conditional:Npnn \__template_if_use_template:n #1 { TF }
92 {
93   \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil \q_stop }
94   \prg_return_true:
95   \prg_return_false:
96 }
```

(End of definition for `__template_if_use_template:nTF`.)

11.3 Saving and recovering property lists

The various property lists for templates have to be shuffled in and out of storage.

The defaults and keytypes are transferred from the scratch property lists to the “proper” lists for the template being created.

```

__template_store_defaults:nn
__template_store_keytypes:nn
__template_store_values:nn
__template_store_vars:nn
97 \cs_new_protected:Npn \__template_store_defaults:nn #1#2
98 {
99   \debug_suspend:
100   \prop_gclear_new:c { \c__template_defaults_root_tl #1 / #2 }
101   \prop_gset_eq:cN { \c__template_defaults_root_tl #1 / #2 }
102   \l__template_values_prop
103   \debug_resume:
104 }
105 \cs_new_protected:Npn \__template_store_keytypes:nn #1#2
106 {
107   \debug_suspend:
108   \prop_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
109   {
110     \msg_info:nnnn { template } { declare-template-interface } {#1} {#2}
111     \prop_gclear:c { \c__template_keytypes_root_tl #1 / #2 }
112   }
113   { \prop_new:c { \c__template_keytypes_root_tl #1 / #2 } }
114   \prop_gset_eq:cN { \c__template_keytypes_root_tl #1 / #2 }
115   \l__template_keytypes_prop
116   \seq_gclear_new:c { \c__template_key_order_root_tl #1 / #2 }
117   \seq_gset_eq:cN { \c__template_key_order_root_tl #1 / #2 }
118   \l__template_key_order_seq
119   \debug_resume:
120 }
121 \cs_new_protected:Npn \__template_store_values:nn #1#2
122 {
123   \debug_suspend:
124   \prop_clear_new:c { \c__template_values_root_tl #1 / #2 }
125   \prop_set_eq:cN { \c__template_values_root_tl #1 / #2 }
126   \l__template_values_prop
127   \debug_resume:
128 }
129 \cs_new_protected:Npn \__template_store_vars:nn #1#2
130 {
131   \debug_suspend:
132   \prop_gclear_new:c { \c__template_vars_root_tl #1 / #2 }
133   \prop_gset_eq:cN { \c__template_vars_root_tl #1 / #2 }
134   \l__template_vars_prop
135   \debug_resume:
136 }

```

(End of definition for __template_store_defaults:nn and others.)

Recovering the stored data for a template is rather less complex than storing it. All that happens is the data is transferred from the permanent to the scratch storage. However, we need to check the scratch storage does exist.

```

__template_recover_defaults:nn
__template_recover_keytypes:nn
__template_recover_values:nn
__template_recover_vars:nn
137 \cs_new_protected:Npn \__template_recover_defaults:nn #1#2
138 {

```

```

139     \prop_if_exist:cTF
140     { \c__template_defaults_root_tl #1 / #2 }
141     {
142     \prop_set_eq:Nc \l__template_values_prop
143     { \c__template_defaults_root_tl #1 / #2 }
144     }
145     { \prop_clear:N \l__template_values_prop }
146   }
147 \cs_new_protected:Npn \__template_recover_keytypes:nn #1#2
148 {
149   \prop_if_exist:cTF
150   { \c__template_keytypes_root_tl #1 / #2 }
151   {
152   \prop_set_eq:Nc \l__template_keytypes_prop
153   { \c__template_keytypes_root_tl #1 / #2 }
154   }
155   { \prop_clear:N \l__template_keytypes_prop }
156   \seq_if_exist:cTF { \c__template_key_order_root_tl #1 / #2 }
157   {
158     \seq_set_eq:Nc \l__template_key_order_seq
159     { \c__template_key_order_root_tl #1 / #2 }
160   }
161   { \seq_clear:N \l__template_key_order_seq }
162 }
163 \cs_new_protected:Npn \__template_recover_values:nn #1#2
164 {
165   \prop_if_exist:cTF
166   { \c__template_values_root_tl #1 / #2 }
167   {
168     \prop_set_eq:Nc \l__template_values_prop
169     { \c__template_values_root_tl #1 / #2 }
170   }
171   { \prop_clear:N \l__template_values_prop }
172 }
173 \cs_new_protected:Npn \__template_recover_vars:nn #1#2
174 {
175   \prop_if_exist:cTF
176   { \c__template_vars_root_tl #1 / #2 }
177   {
178     \prop_set_eq:Nc \l__template_vars_prop
179     { \c__template_vars_root_tl #1 / #2 }
180   }
181   { \prop_clear:N \l__template_vars_prop }
182 }

```

(End of definition for __template_recover_defaults:nn and others.)

11.4 Creating new template types

Although the type is the “top level” of the template system, it is actually very easy to implement. All that happens is that the number of arguments required is recorded, indexed by the name of the type.

```

183 \cs_new_protected:Npn \__template_define_type:nn #1#2
184 {

```



```

185     \prop_if_in:NnTF \g__template_type_prop {#1}
186     { \msg_error:nnn { template } { type-already-defined } {#1} }
187     { \__template_declare_type:nn {#1} {#2} }
188   }
189 \cs_new_protected:Npn \__template_declare_type:nn #1#2
190 {
191   \int_set:Nn \l__template_tmp_int {#2}
192   \int_compare:nTF { 0 <= \l__template_tmp_int <= 9 }
193   {
194     \msg_info:nnnV { template } { declare-type }
195     {#1} \l__template_tmp_int
196     \prop_gput:NnV \g__template_type_prop {#1}
197     \l__template_tmp_int
198   }
199   {
200     \msg_error:nnnV { template } { bad-number-of-arguments }
201     {#1} \l__template_tmp_int
202   }
203 }

```

(End of definition for __template_define_type:nn and __template_declare_type:nn.)

11.5 Design part of template declaration

The “design” part of a template declaration defines the general behaviour of each key, and possibly a default value. However, it does not include the implementation. This means that what happens here is the two properties are saved to appropriate lists, which can then be used later to recover the information when implementing the keys.

`__template_declare_template_keys:nnnn`

The main function for the “design” part of creating a template starts by checking that the type exists and that the number of arguments required agree. If that is all fine, then the two storage areas for defaults and keytypes are initialised. The mechanism is then set up for the `l3keys` module to actually parse the keys. Finally, the code hands off to the storage routine to save the parsed information properly.

```

204 \cs_new_protected:Npn \__template_declare_template_keys:nnnn #1#2#3#4
205 {
206   \__template_execute_if_type_exist:nT {#1}
207   {
208     \__template_execute_if_arg_agree:nnT {#1} {#3}
209     {
210       \prop_clear:N \l__template_values_prop
211       \prop_clear:N \l__template_keytypes_prop
212       \seq_clear:N \l__template_key_order_seq
213       \keyval_parse:NNn
214       \__template_parse_keys_elt:n \__template_parse_keys_elt:nn {#4}
215       \__template_store_defaults:nn {#1} {#2}
216       \__template_store_keytypes:nn {#1} {#2}
217     }
218   }
219 }

```

(End of definition for __template_declare_template_keys:nnnn.)

`_template_parse_keys_elt:n` Processing the key part of the key–value pair is always carried out using this function, even if a value was found. First, the key name is separated from the keytype, and if necessary the keytype is separated into two parts. This information is then used to check that the keytype is valid, before storing the keytype (plus argument if necessary) as a property of the key name. The key name is also stored (in braces) in the token list to record the order the keys are defined in.

```

220 \\cs_new_protected:Npn \\_template_parse_keys_elt:n #1
221 {
222   \\_template_split_keytype:n {#1}
223   \\bool_if:NF \\l__template_error_bool
224   {
225     \\_template_execute_if_keytype_exist:VT \\l__template_keytype_tl
226     {
227       \\seq_map_function:NN \\c__template_keytypes_arg_seq
228       \\_template_parse_keys_elt_aux:n
229       \\bool_if:NF \\l__template_error_bool
230       {
231         \\seq_if_in:NoTF \\l__template_key_order_seq
232         \\l__template_key_name_tl
233         {
234           \\msg_error:nnV { template } { duplicate-key-interface }
235           \\l__template_key_name_tl
236         }
237         { \\_template_parse_keys_elt_aux: }
238       }
239     }
240   }
241 }
242 \\cs_new_protected:Npn \\_template_parse_keys_elt_aux:n #1
243 {
244   \\str_if_eq:VnT \\l__template_keytype_tl {#1}
245   {
246     \\tl_if_empty:NT \\l__template_keytype_arg_tl
247     {
248       \\msg_error:nnn { template } { keytype-requires-argument } {#1}
249       \\bool_set_true:N \\l__template_error_bool
250       \\seq_map_break:
251     }
252   }
253 }
254 \\cs_new_protected:Npn \\_template_parse_keys_elt_aux:
255 {
256   \\tl_set:Ne \\l__template_tmp_tl
257   {
258     \\l__template_keytype_tl
259     \\tl_if_empty:NF \\l__template_keytype_arg_tl
260     { { \\l__template_keytype_arg_tl } }
261   }
262   \\prop_put:NVV \\l__template_keytypes_prop \\l__template_key_name_tl
263   \\l__template_tmp_tl
264   \\seq_put_right:NV \\l__template_key_order_seq \\l__template_key_name_tl
265   \\str_if_eq:VnT \\l__template_keytype_tl { choice }
266   {

```

```

267         \clist_if_in:NnT \l__template_keytype_arg_tl { unknown }
268         { \msg_error:nn { template } { choice-unknown-reserved } }
269     }
270 }

```

(End of definition for `__template_parse_keys_elt:n`, `__template_parse_keys_elt_aux:n`, and `__template_parse_keys_elt_aux:.`)

`__template_parse_keys_elt:nn` For keys which have a default, the keytype and key name are first separated out by the `__template_parse_keys_elt:n` routine, before storing the default value in the scratch property list.

```

271 \cs_new_protected:Npn \__template_parse_keys_elt:nn #1#2
272 {
273     \__template_parse_keys_elt:n {#1}
274     \use:c { __template_store_value_ \l__template_keytype_tl :n } {#2}
275 }

```

(End of definition for `__template_parse_keys_elt:nn`.)

`__template_split_keytype:n` The keytype and key name should be separated by `:`. As the definition might be given inside or outside of a code block, the category code of colons is standardised. After that, the standard delimited argument method is used to separate the two parts.

`__template_split_keytype_aux:w`

```

276 \cs_new_protected:Npe \__template_split_keytype:n #1
277 {
278     \exp_not:N \bool_set_false:N \exp_not:N \l__template_error_bool
279     \tl_set:Nn \exp_not:N \l__template_tmp_tl {#1}
280     \tl_replace_all:Nnn \exp_not:N \l__template_tmp_tl { : } { \token_to_str:N : }
281     \tl_if_in:VnTF \exp_not:N \l__template_tmp_tl { \token_to_str:N : }
282     {
283         \exp_not:n
284         {
285             \tl_clear:N \l__template_key_name_tl
286             \exp_after:wN \__template_split_keytype_aux:w
287             \l__template_tmp_tl \s__template_stop
288         }
289     }
290     {
291         \exp_not:N \bool_set_true:N \exp_not:N \l__template_error_bool
292         \msg_error:nnn { template } { missing-keytype } {#1}
293     }
294 }
295 \use:e
296 {
297     \cs_new_protected:Npn \exp_not:N \__template_split_keytype_aux:w
298     #1 \token_to_str:N : #2 \s__template_stop
299     {
300         \tl_put_right:Ne \exp_not:N \l__template_key_name_tl
301         {
302             \exp_not:N \tl_trim_spaces:e
303             { \exp_not:N \tl_to_str:n {#1} }
304         }
305         \tl_if_in:nnTF {#2} { \token_to_str:N : }
306         {
307             \tl_put_right:Nn \exp_not:N \l__template_key_name_tl

```

```

308         { \token_to_str:N : }
309         \exp_not:N \__template_split_keytype_aux:w #2 \s__template_stop
310     }
311     {
312         \exp_not:N \tl_if_empty:NTF \exp_not:N \l__template_key_name_tl
313         {
314             \msg_error:nnn { template } { empty-key-name }
315             { \token_to_str:N : #2 }
316         }
317         { \exp_not:N \__template_split_keytype_arg:n {#2} }
318     }
319 }
320 }

```

(End of definition for __template_split_keytype:n and __template_split_keytype_aux:w.)

__template_split_keytype_arg:n
 __template_split_keytype_arg:V
 __template_split_keytype_arg_aux:n
 __template_split_keytype_arg_aux:w

The second stage of sorting out the keytype is to check for an argument. As there is no convenient delimiting token to look for, a check is made instead for each possible text value for the keytype. To keep things faster, this only involves the keytypes that need an argument. If a match is made, then a check is also needed to see that it is at the start of the keytype information. All being well, the split can then be applied. Any non-matching keytypes are assumed to be “correct” as given, and are left alone (this is checked by other code).

```

321 \cs_new_protected:Npn \__template_split_keytype_arg:n #1
322 {
323     \tl_set:Ne \l__template_keytype_tl { \tl_trim_spaces:n {#1} }
324     \tl_clear:N \l__template_keytype_arg_tl
325     \cs_set_protected:Npn \__template_split_keytype_arg_aux:n ##1
326     {
327         \tl_if_in:nnT {#1} {###1}
328         {
329             \cs_set:Npn \__template_split_keytype_arg_aux:w
330             ###1 ##1 ###2 \s__template_stop
331             {
332                 \tl_if_blank:nT {###1}
333                 {
334                     \tl_set:Ne \l__template_keytype_tl
335                     { \tl_trim_spaces:n {##1} }
336                     \tl_if_blank:nF {###2}
337                     {
338                         \tl_set:Ne \l__template_keytype_arg_tl
339                         { \use:n {###2} }
340                     }
341                     \seq_map_break:
342                 }
343             }
344             \__template_split_keytype_arg_aux:w #1 \s__template_stop
345         }
346     }
347     \seq_map_function:NN \c__template_keytypes_arg_seq
348     \__template_split_keytype_arg_aux:n
349 }
350 \cs_generate_variant:Nn \__template_split_keytype_arg:n { V }
351 \cs_new:Npn \__template_split_keytype_arg_aux:n #1 { }

```

```
352 \cs_new:Npn \__template_split_keytype_arg_aux:w #1 \s__template_stop { }
```

(End of definition for `__template_split_keytype_arg:n`, `__template_split_keytype_arg_aux:n`, and `__template_split_keytype_arg_aux:w`.)

11.5.1 Storing values

As `ltemplates` pre-processes key values for efficiency reasons, there is a need to convert the values given as defaults into “ready to use” data. The same general idea is true when an instance is declared. However, assignments are not made until an instance is used, and so there has to be some intermediate storage. Furthermore, the ability to delay evaluation of results is needed. To achieve these aims, a series of “process and store” functions are defined here.

All of the information about the key (the key name and the keytype) is already stored as variables. The same property list is always used to store the data, meaning that the only argument required is the value to be processed and potentially stored.

```
\__template_store_value_boolean:n
```

```
353 \cs_new_protected:Npn \__template_store_value_boolean:n #1
354 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }
```

(End of definition for `__template_store_value_boolean:n`.)

```
\__template_store_value:n
```

With no need to worry about delayed evaluation, these keytypes all just store the input directly.

```
\__template_store_value_choice:n
\__template_store_value_function:n
\__template_store_value_instance:n
355 \cs_new_protected:Npn \__template_store_value:n #1
356 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }
357 \cs_new_eq:NN \__template_store_value_choice:n \__template_store_value:n
358 \cs_new_eq:NN \__template_store_value_function:n \__template_store_value:n
359 \cs_new_eq:NN \__template_store_value_instance:n \__template_store_value:n
```

(End of definition for `__template_store_value:n` and others.)

```
\__template_store_value_aux:Nn
```

Storing values in `\l__template_values_prop` is in most cases the same.

```
\__template_store_value_integer:n
\__template_store_value_length:n
\__template_store_value_muskip:n
\__template_store_value_real:n
\__template_store_value_skip:n
\__template_store_value_tokenlist:n
\__template_store_value_commalist:n
360 \cs_new_protected:Npn \__template_store_value_aux:Nn #1#2
361 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#2} }
362 \cs_new_protected:Npn \__template_store_value_integer:n
363 { \__template_store_value_aux:Nn \int_eval:n }
364 \cs_new_protected:Npn \__template_store_value_length:n
365 { \__template_store_value_aux:Nn \dim_eval:n }
366 \cs_new_protected:Npn \__template_store_value_muskip:n
367 { \__template_store_value_aux:Nn \muskip_eval:n }
368 \cs_new_protected:Npn \__template_store_value_real:n
369 { \__template_store_value_aux:Nn \fp_eval:n }
370 \cs_new_protected:Npn \__template_store_value_skip:n
371 { \__template_store_value_aux:Nn \skip_eval:n }
372 \cs_new_protected:Npn \__template_store_value_tokenlist:n
373 { \__template_store_value_aux:Nn \use:n }
374 \cs_new_eq:NN \__template_store_value_commalist:n \__template_store_value_tokenlist:n
```

(End of definition for `__template_store_value_aux:Nn` and others.)

11.6 Implementation part of template declaration

`__template_declare_template_code:nnnn`
`__template_declare_template_code:nnnn`

The main function for implementing a template starts with a couple of simple checks to make sure that there are no obvious mistakes: the number of arguments must agree and the template keys must have been declared.

```

375 \cs_new_protected:Npn __template_declare_template_code:nnnn #1#2#3#4#5
376 {
377   __template_execute_if_type_exist:nT {#1}
378   {
379     __template_execute_if_arg_agree:nnT {#1} {#3}
380     {
381       __template_if_keys_exist:nnT {#1} {#2}
382       {
383         __template_store_key_implementation:nnn {#1} {#2} {#4}
384         \str_if_in:nnTF {#5} { AssignTemplateKeys }
385         {
386           \regex_match:nnTF { \c { AssignTemplateKeys } } {#5}
387           { __template_declare_template_code:nnnn {#1} {#2} {#3} {#5} }
388           {
389             __template_declare_template_code:nnnn
390             {#1} {#2} {#3} { \AssignTemplateKeys #5 }
391           }
392         }
393       }
394       __template_declare_template_code:nnnn
395       {#1} {#2} {#3} { \AssignTemplateKeys #5 }
396     }
397   }
398 }
399 }
400 }
401 \cs_new_protected:Npn __template_declare_template_code:nnnn #1#2#3#4
402 {
403   \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
404   { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
405   \cs_generate_from_arg_count:cNnn
406   { \c__template_code_root_tl #1 / #2 }
407   \cs_gset_protected:Npn {#3} {#4}
408 }

```

(End of definition for `__template_declare_template_code:nnnn` and `__template_declare_template_code:nnnn`.)

`__template_store_key_implementation:nnn`

Actually storing the implementation part of a template is quite easy as it only requires the list of keys given to be turned into a property list. There is also some error-checking to do, hence the need to have the list of defined keytypes available. In certain cases (when choices are involved) parsing the key results in changes to the default values. That is why they are loaded and then saved again.

```

409 \cs_new_protected:Npn __template_store_key_implementation:nnn #1#2#3
410 {
411   __template_recover_defaults:nn {#1} {#2}
412   __template_recover_keytypes:nn {#1} {#2}
413   \prop_clear:N \l__template_vars_prop
414   \keyval_parse:nnn

```

```

415     { \__template_parse_vars_elt:n } { \__template_parse_vars_elt:nnn { #1 / #2 } } {#3}
416 \__template_store_vars:nn {#1} {#2}
417 \prop_map_inline:Nn \l__template_keytypes_prop
418   { \msg_error:nnnn { template } { key-not-implemented } {##1} {#2} {#1} }
419 }

```

(End of definition for __template_store_key_implementation:nnn.)

__template_parse_vars_elt:n At the implementation stage, every key must have a value given. So this is an error function.

```

420 \cs_new_protected:Npn \__template_parse_vars_elt:n #1
421   { \msg_error:nnn { template } { key-no-variable } {#1} }

```

(End of definition for __template_parse_vars_elt:n.)

__template_parse_vars_elt:nnn The actual storage part here is very simple: the storage bin name is placed into the property list. At the same time, a comparison is made with the keytypes defined earlier: if there is a mismatch then an error is raised.

```

422 \cs_new_protected:Npn \__template_parse_vars_elt:nnn #1#2#3
423   {
424     \tl_set:Ne \l__template_key_name_tl
425       { \tl_trim_spaces:e { \tl_to_str:n {#2} } } }
426     \prop_get:NVNTF \l__template_keytypes_prop
427       \l__template_key_name_tl
428       \l__template_keytype_tl
429     {
430       \__template_split_keytype_arg:V \l__template_keytype_tl
431       \__template_parse_vars_elt_aux:nn {#1} {#3}
432       \prop_remove:NV \l__template_keytypes_prop \l__template_key_name_tl
433     }
434     { \msg_error:nnn { template } { unknown-key } {#2} }
435   }

```

Split off any leading global and they look for the way to implement.

```

436 \cs_new_protected:Npn \__template_parse_vars_elt_aux:nn #1#2
437   {
438     \__template_parse_vars_elt_aux:nw {#1} #2 global global \s__template_stop
439   }
440 \cs_new_protected:Npn \__template_parse_vars_elt_aux:nw
441   #1#2 global #3 global #4 \s__template_stop
442   {
443     \tl_if_blank:nTF {#4}
444     { \__template_parse_vars_elt_aux:nnn {#1} { } {#2} }
445     {
446       \tl_if_blank:nTF {#2}
447       {
448         \__template_parse_vars_elt_aux:nne
449           {#1} { global } { \tl_trim_spaces:n {#3} } }
450       }
451     { \msg_error:nnn { template } { bad-variable } { #2 global #3 } }
452   }
453 }
454 \cs_new_protected:Npn \__template_parse_vars_elt_aux:nnn #1#2#3
455   {
456     \str_case:VnF \l__template_keytype_tl

```

```

457 {
458 { choice } { \_template_implement_choices:nn {#1} {#3} }
459 { function }
460 {
461   \cs_if_exist:NF #3
462   { \cs_new:Npn #3 { } }
463   \_template_parse_vars_elt_key:nn {#1}
464   {
465     .code:n =
466     {
467       \cs_generate_from_arg_count:NNnn
468       \exp_not:N #3
469       \exp_not:c
470       { cs_ \str_if_eq:nnT {#1} { global } { g } set:Npn }
471       { \exp_not:V \l__template_keytype_arg_tl }
472       {##1}
473     }
474   }
475   \prop_put:NVn \l__template_vars_prop
476   \l__template_key_name_tl {#2#3}
477 }
478 { instance }
479 {
480   \_template_parse_vars_elt_key:nn {#1}
481   {
482     .code:n =
483     {
484       \exp_not:c
485       { cs_ \str_if_eq:nnT {#1} { global } { g } set:Npn }
486       \exp_not:N #3 { \UseInstance {##1} }
487     }
488   }
489   \prop_put:NVn \l__template_vars_prop
490   \l__template_key_name_tl {#2#3}
491 }
492 }
493 {
494   \tl_if_single:nTF {#3}
495   {
496     \cs_if_exist:NF #3
497     { \use:c { \_template_map_var_type: _new:N } #3 }
498     \_template_parse_vars_elt_key:nn {#1}
499     {
500       . \_template_map_var_type:
501       - \str_if_eq:nnT {#1} { global } { g } set:N
502       = \exp_not:N #3
503     }
504     \prop_put:NVn \l__template_vars_prop
505     \l__template_key_name_tl {#2#3}
506   }
507   { \msg_error:nnn { template } { bad-variable } {#2#3} }
508 }
509 }
510 \cs_generate_variant:Nn \_template_parse_vars_elt_aux:nnn { nne }

```



```

511 \cs_new_protected:Npn \__template_parse_vars_elt_key:nn #1#2
512 {
513   \keys_define:ne { template / #1 }
514   { \l__template_key_name_tl #2 }
515 }

```

(End of definition for __template_parse_vars_elt:nnn and others.)

__template_map_var_type: Turn a “friendly” variable type into an expl3 one.

```

516 \cs_new:Npn \__template_map_var_type:
517 {
518   \str_case:Vn \l__template_keytype_tl
519   {
520     { boolean } { bool }
521     { commalist } { clist }
522     { integer } { int }
523     { length } { dim }
524     { muskip } { muskip }
525     { real } { fp }
526     { skip } { skip }
527     { tokenlist } { tl }
528   }
529 }

```

(End of definition for __template_map_var_type:.)

__template_implement_choices:nn __template_implement_choices_default: Implementing choices requires a second key–value loop. So after a little set-up, the standard parser is called.

```

530 \cs_new_protected:Npn \__template_implement_choices:nn #1#2
531 {
532   \clist_set:NV \l__template_tmp_clist \l__template_keytype_arg_tl
533   \prop_put:NVn \l__template_vars_prop \l__template_key_name_tl { }
534   \keys_define:ne { template / #1 } { \l__template_key_name_tl .choice: }
535   \keyval_parse:nnn
536   { \__template_implement_choice_elt:n }
537   { \__template_implement_choice_elt:nnn {#1} }
538   {#2}
539   \prop_get:NVNT \l__template_values_prop \l__template_key_name_tl
540   \l__template_tmp_tl
541   { \__template_implement_choices_default: }
542   \clist_if_empty:NF \l__template_tmp_clist
543   {
544     \clist_map_inline:Nn \l__template_tmp_clist
545     { \msg_error:nnn { template } { choice-not-implemented } {##1} }
546   }
547 }

```

A sanity check for the default value, so that an error is raised now and not when converting to assignments.

```

548 \cs_new_protected:Npn \__template_implement_choices_default:
549 {
550   \tl_set:Ne \l__template_tmp_tl
551   { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
552   \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
553   {

```

```

554     \tl_set:Ne \l__template_tmp_tl
555     { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
556     \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
557     {
558         \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
559         \l__template_tmp_tl
560         \__template_split_keytype_arg:V \l__template_tmp_tl
561         \prop_get:NVN \l__template_values_prop \l__template_key_name_tl
562         \l__template_tmp_tl
563         \msg_error:nnVV { template } { unknown-default-choice }
564         \l__template_key_name_tl
565         \l__template_key_name_tl
566     }
567 }
568 }

```

(End of definition for `__template_implement_choices:nn` and `__template_implement_choices_default:.`)

```

\__template_implement_choice_elt:nnn
\__template_implement_choice_elt_aux:nnn
\__template_implement_choice_elt_aux:n
\__template_implement_choice_elt:n

```

The actual storage of the implementation of a choice is mainly about error checking. The code here ensures that all choices have to have been declared, apart from the special **unknown** choice, which must come last. The code for each choice is stored along with the key name in the variables property list.

```

569 \cs_new_protected:Npn \__template_implement_choice_elt:nnn #1#2#3
570 {
571     \clist_if_empty:NTF \l__template_tmp_clist
572     {
573         \str_if_eq:nnTF {#2} { unknown }
574         { \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3} }
575         { \__template_implement_choice_elt_aux:n {#2} }
576     }
577     {
578         \clist_if_in:NnTF \l__template_tmp_clist {#2}
579         {
580             \clist_remove_all:Nn \l__template_tmp_clist {#2}
581             \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3}
582         }
583         { \__template_implement_choice_elt_aux:n {#2} }
584     }
585 }
586 \cs_new_protected:Npn \__template_implement_choice_elt_aux:n #1
587 {
588     \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
589     \l__template_tmp_tl
590     \__template_split_keytype_arg:V \l__template_tmp_tl
591     \msg_error:nnVn { template } { unknown-choice } \l__template_key_name_tl {#1}
592 }
593 \cs_new_protected:Npn \__template_implement_choice_elt_aux:nnn #1#2#3
594 {
595     \keys_define:ne { template / #1 }
596     { \l__template_key_name_tl / #2 .code:n = { \exp_not:n {#3} } }
597     \tl_set:Ne \l__template_tmp_tl
598     { \l__template_key_name_tl \c_space_tl #2 }
599     \prop_put:NVn \l__template_vars_prop \l__template_tmp_tl {#3}

```

```

600 }
601 \cs_new_protected:Npn \__template_implement_choice_elt:n #1
602 {
603   \msg_error:nnVn { template } { choice-requires-code }
604   \l__template_key_name_tl {#1}
605 }

```

(End of definition for __template_implement_choice_elt:nnn and others.)

11.7 Editing template defaults

`__template_edit_defaults:nnn` Editing the template defaults means getting the values back out of the store, then parsing the list of new values before putting the updated list back into storage.

```

606 \cs_new_protected:Npn \__template_edit_defaults:nnn #1#2#3
607 {
608   \__template_if_keys_exist:nnT {#1} {#2}
609   {
610     \__template_recover_defaults:nn {#1} {#2}
611     \__template_parse_values:nnn {#1} {#2} {#3}
612     \__template_store_defaults:nn {#1} {#2}
613   }
614 }

```

(End of definition for __template_edit_defaults:nnn.)

`__template_parse_values:nnn` The routine to parse values is the same for both editing a template and setting up an instance. So the code here does only the minimum necessary for reading the values.

```

615 \cs_new_protected:Npn \__template_parse_values:nnn #1#2#3
616 {
617   \__template_recover_keytypes:nn {#1} {#2}
618   \keyval_parse:NNn
619   \__template_parse_values_elt:n \__template_parse_values_elt:n {#3}
620 }

```

(End of definition for __template_parse_values:nnn.)

`__template_parse_values_elt:n` Every key needs a value, so this is just an error routine.

```

621 \cs_new_protected:Npn \__template_parse_values_elt:n #1
622 {
623   \bool_set_true:N \l__template_error_bool
624   \msg_error:nnn { template } { key-no-value } {#1}
625 }

```

(End of definition for __template_parse_values_elt:n.)

`__template_parse_values_elt:n` To store the value, find the keytype then call the saving function. These need the current key name, stored in `\l__template_key_name_tl`.

```

626 \cs_new_protected:Npn \__template_parse_values_elt:n #1#2
627 {
628   \use:e
629   {
630     \__template_parse_values_elt_aux:w
631     \tl_trim_spaces:e { \tl_to_str:n { #1 : n : } }
632     \exp_not:N \q_stop

```

```

633     }
634     \prop_get:NVNTF \l__template_keytypes_prop \l__template_key_name_tl
635     \l__template_tmp_tl
636     { \__template_parse_values_elt_aux:n {#2} }
637     { \msg_error:nnV { template } { unknown-key } \l__template_key_name_tl }
638   }
639   \use:e
640   {
641     \cs_new_protected:Npn \exp_not:N \__template_parse_values_elt_aux:w
642     #1 \token_to_str:N : #2 \token_to_str:N : #3 \exp_not:N \q_stop
643   }
644   {
645     \tl_set:Nn \l__template_key_name_tl {#1}
646     \str_set:Nn \l__template_value_exp_str {#2}
647   }
648   \cs_new_protected:Npn \__template_parse_values_elt_aux:n #1
649   {
650     \__template_split_keytype_arg:V \l__template_tmp_tl
651     \cs_if_exist_use:cF { __template_parse_values_exp: \l__template_value_exp_str }
652     {
653       \msg_error:nnV { template } { unknown-expansion } \l__template_value_exp_str
654       \use_none:n
655     }
656     {#1}
657   }
658   \cs_new_protected:Npn \__template_parse_values_exp:n #1
659   { \use:c { __template_store_value \l__template_keytype_tl :n } {#1} }
660   \cs_generate_variant:Nn \__template_parse_values_exp:n { o , V , v , e }
661   \cs_new_eq:NN \__template_parse_values_exp:N \__template_parse_values_exp:n
662   \cs_generate_variant:Nn \__template_parse_values_exp:N { c }

```

(End of definition for `__template_parse_values_elt:nn` and others.)

`__template_template_set_eq:nnn` To copy a template, each of the lists plus the code has to be copied across. To keep this independent of the list storage system, it is all done with two-part shuffles.

```

663   \cs_new_protected:Npn \__template_template_set_eq:nnn #1#2#3
664   {
665     \__template_recover_defaults:nn {#1} {#3}
666     \__template_store_defaults:nn {#1} {#2}
667     \__template_recover_keytypes:nn {#1} {#3}
668     \__template_store_keytypes:nn {#1} {#2}
669     \__template_recover_vars:nn {#1} {#3}
670     \__template_store_vars:nn {#1} {#2}
671     \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
672     { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
673     \cs_gset_eq:cc { \c__template_code_root_tl #1 / #2 }
674     { \c__template_code_root_tl #1 / #3 }
675   }

```

(End of definition for `__template_template_set_eq:nnn`.)

11.8 Creating instances of templates

`__template_declare_instance:nnnn`
`__template_declare_instance_aux:nnnn` Making an instance has two distinct parts. First, the keys given are parsed to transfer the values into the structured data format used internally. This allows the default and

given values to be combined with no repetition. In the second step, the structured data is converted to pre-defined variable assignments, and these are stored in the function for the instance.

```

676 \cs_new_protected:Npn \__template_declare_instance:nnnn #1#2#3#4
677 {
678   \__template_execute_if_code_exist:nnT {#1} {#2}
679   {
680     \__template_recover_defaults:nn {#1} {#2}
681     \__template_recover_vars:nn {#1} {#2}
682     \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
683   }
684 }
685 \cs_new_protected:Npn \__template_declare_instance_aux:nnnn #1#2#3#4
686 {
687   \bool_set_false:N \l__template_error_bool
688   \__template_parse_values:nnn {#1} {#2} {#4}
689   \bool_if:NF \l__template_error_bool
690   {
691     \prop_put:Nnn \l__template_values_prop { from-template } {#2}
692     \__template_store_values:nn {#1} {#3}
693     \__template_convert_to_assignments:
694     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #3 }
695     { \msg_info:nnnn { template } { declare-instance } {#3} {#1} }
696     \cs_set_protected:cpe { \c__template_instances_root_tl #1 / #3 }
697     {
698       \exp_not:N \__template_assignments_push:n
699       { \exp_not:V \l__template_assignments_tl }
700       \exp_not:c { \c__template_code_root_tl #1 / #2 }
701     }
702   }
703 }

```

(End of definition for `__template_declare_instance:nnnn` and `__template_declare_instance_aux:nnnn`.)

`__template_instance_set_eq:nnn` Copy-paste an instance.

```

704 \cs_new_protected:Npn \__template_instance_set_eq:nnn #1#2#3
705 {
706   \__template_if_instance_exist:nnTF {#1} {#3}
707   {
708     \__template_recover_values:nn {#1} {#3}
709     \__template_store_values:nn {#1} {#2}
710     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #2 }
711     { \msg_info:nnnn { template } { declare-instance } {#2} {#1} }
712     \cs_set_eq:cc { \c__template_instances_root_tl #1 / #2 }
713     { \c__template_instances_root_tl #1 / #3 }
714   }
715   { \msg_error:nnnn { template } { unknown-instance } {#1} {#3} }
716 }

```

(End of definition for `__template_instance_set_eq:nnn`.)

`__template_edit_instance:nnn` Editing an instance is almost identical to declaring one. The only variation is the source of the values to use. When editing, they are recovered from the previous instance run.
`__template_edit_instance_aux:nnnn`
`__template_edit_instance_aux:nVnnn`

```

717 \cs_new_protected:Npn \__template_edit_instance:nnn #1#2#3
718 {
719   \__template_if_instance_exist:nnTF {#1} {#2}
720   {
721     \__template_recover_values:nn {#1} {#2}
722     \prop_get:NnN \l__template_values_prop { from~template }
723     \l__template_tmp_tl
724     \__template_edit_instance_aux:nVnn
725     {#1} \l__template_tmp_tl {#2} {#3}
726   }
727   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
728 }
729 \cs_new_protected:Npn \__template_edit_instance_aux:nnnn #1#2#3#4
730 {
731   \__template_recover_vars:nn {#1} {#2}
732   \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
733 }
734 \cs_generate_variant:Nn \__template_edit_instance_aux:nnnn { nV }

```

(End of definition for __template_edit_instance:nnn and __template_edit_instance_aux:nnnn.)

```

\__template_convert_to_assignments:
\__template_convert_to_assignments_aux:n
\__template_convert_to_assignments_aux:nn
\__template_convert_to_assignments_aux:nV

```

The idea on converting to a set of assignments is to loop over each key, so that the loop order follows the declaration order of the keys. This is done using a sequence as property lists are not “ordered”.

```

735 \cs_new_protected:Npn \__template_convert_to_assignments:
736 {
737   \tl_clear:N \l__template_assignments_tl
738   \seq_map_function:NN \l__template_key_order_seq
739   \__template_convert_to_assignments_aux:n
740 }
741 \cs_new_protected:Npn \__template_convert_to_assignments_aux:n #1
742 {
743   \prop_get:NnN \l__template_keytypes_prop {#1} \l__template_tmp_tl
744   \__template_convert_to_assignments_aux:nV {#1} \l__template_tmp_tl
745 }

```

The second auxiliary function actually does the work. The arguments here are the key name (#1) and the keytype (#2). From those, the value to assign and the name of the appropriate variable are recovered. A bit of work is then needed to sort out keytypes with arguments (for example instances), and to look for global assignments. Once that is done, a hand-off can be made to the handler for the relevant keytype.

```

746 \cs_new_protected:Npn \__template_convert_to_assignments_aux:nn #1#2
747 {
748   \prop_get:NnNT \l__template_values_prop {#1} \l__template_value_tl
749   {
750     \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_var_tl
751     {
752       \__template_split_keytype_arg:n {#2}
753       \str_if_eq:VnF \l__template_keytype_tl { choice }
754       {
755         \str_if_eq:VnF \l__template_keytype_tl { code }
756         { \__template_find_global: }
757       }
758     }
759     \tl_set:Nn \l__template_key_name_tl {#1}

```

```

759         \cs_if_exist_use:cF { __template_assign_ \l__template_keytype_tl : }
760         { __template_assign_variable: }
761     }
762     { \msg_error:nnn { template } { unknown-attribute } {#1} }
763 }
764 }
765 \cs_generate_variant:Nn \__template_convert_to_assignments_aux:nn { nV }

```

(End of definition for `__template_convert_to_assignments:`, `__template_convert_to_assignments_aux:n`, and `__template_convert_to_assignments_aux:nn`.)

`__template_find_global:` Global assignments should have the phrase `global` at the front. This is pretty easy to find: no other error checking, though.

`__template_find_global_aux:w`

```

766 \cs_new_protected:Npn \__template_find_global:
767 {
768     \bool_set_false:N \l__template_global_bool
769     \tl_if_in:onT \l__template_var_tl { global }
770     {
771         \exp_after:wN \__template_find_global_aux:w \l__template_var_tl \s__template_stop
772     }
773 }
774 \cs_new_protected:Npn \__template_find_global_aux:w #1 global #2 \s__template_stop
775 {
776     \tl_set:Nn \l__template_var_tl {#2}
777     \bool_set_true:N \l__template_global_bool
778 }

```

(End of definition for `__template_find_global:` and `__template_find_global_aux:w`.)

11.9 Using templates directly

`__template_use_template:nnn` Directly use a template with a particular parameter setting. This is also picked up if used in a nested fashion inside a parameter list. The idea is essentially the same as creating an instance, just with no saving of the result.

```

779 \cs_new_protected:Npn \__template_use_template:nnn #1#2#3
780 {
781     \__template_execute_if_code_exist:nnT {#1} {#2}
782     {
783         \__template_recover_defaults:nn {#1} {#2}
784         \__template_recover_vars:nn {#1} {#2}
785         \__template_parse_values:nnn {#1} {#2} {#3}
786         \__template_convert_to_assignments:
787         \use:c { \c__template_code_root_tl #1 / #2 }
788     }
789 }

```

(End of definition for `__template_use_template:nnn`.)

11.10 Assigning values to variables

`__template_assign_boolean:` Setting a Boolean value is slightly different to everything else as the value can be used to work out which `set` function to call. As long as there is no need to recover things from another variable, everything is pretty easy. If there is, then we need to allow for

the fact that the recovered value here will *not* be expandable, so needs to be converted to something that is.

```

790 \cs_new_protected:Npn \__template_assign_boolean:
791 {
792   \bool_if:NTF \l__template_global_bool
793     { \__template_assign_boolean_aux:n { bool_gset } }
794     { \__template_assign_boolean_aux:n { bool_set } }
795 }
796 \cs_new_protected:Npn \__template_assign_boolean_aux:n #1
797 {
798   \__template_if_key_value:VTF \l__template_value_tl
799   {
800     \__template_key_to_value:
801     \tl_put_right:Ne \l__template_assignments_tl
802     {
803       \exp_not:c { #1 _eq:NN }
804       \exp_not:V \l__template_var_tl
805       \exp_not:V \l__template_value_tl
806     }
807   }
808   {
809     \tl_put_right:Ne \l__template_assignments_tl
810     {
811       \exp_not:c { #1 _ \l__template_value_tl :N }
812       \exp_not:V \l__template_var_tl
813     }
814   }
815 }

```

(End of definition for __template_assign_boolean: and __template_assign_boolean_aux:n.)

__template_assign_choice: The idea here is to find either the choice as-given or else the special unknown choice, and to copy the appropriate code across.

```

\__template_assign_choice_aux:nF
\__template_assign_choice_aux:eF
816 \cs_new_protected:Npn \__template_assign_choice:
817 {
818   \__template_assign_choice_aux:eF
819   { \l__template_key_name_tl \c_space_tl \l__template_value_tl }
820   {
821     \__template_assign_choice_aux:eF
822     { \l__template_key_name_tl \c_space_tl unknown }
823     {
824       \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
825       \l__template_tmp_tl
826       \__template_split_keytype_arg:V \l__template_tmp_tl
827       \msg_error:nnVV { template } { unknown-choice }
828       \l__template_key_name_tl
829       \l__template_value_tl
830     }
831   }
832 }
833 \cs_new_protected:Npn \__template_assign_choice_aux:nF #1
834 {
835   \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_tmp_tl
836   { \tl_put_right:NV \l__template_assignments_tl \l__template_tmp_tl }

```



```

837 }
838 \cs_generate_variant:Nn \__template_assign_choice_aux:nF { e }

```

(End of definition for __template_assign_choice: and __template_assign_choice_aux:nF.)

__template_assign_function: This looks a bit messy but is only actually one function.

```

\__template_assign_function_aux:N
839 \cs_new_protected:Npn \__template_assign_function:
840 {
841   \bool_if:NTF \l__template_global_bool
842     { \__template_assign_function_aux:N \cs_gset:Npn }
843     { \__template_assign_function_aux:N \cs_set:Npn }
844 }
845 \cs_new_protected:Npn \__template_assign_function_aux:N #1
846 {
847   \tl_put_right:Ne \l__template_assignments_tl
848   {
849     \cs_generate_from_arg_count:NNnn
850     \exp_not:V \l__template_var_tl
851     \exp_not:N #1
852     { \exp_not:V \l__template_keytype_arg_tl }
853     { \exp_not:V \l__template_value_tl }
854   }
855 }

```

(End of definition for __template_assign_function: and __template_assign_function_aux:N.)

__template_assign_instance: Using an instance means adding the appropriate function creation to the tl. No checks are made at this stage, so if the instance is not valid then errors will arise later.

```

\__template_assign_instance_aux:N
856 \cs_new_protected:Npn \__template_assign_instance:
857 {
858   \bool_if:NTF \l__template_global_bool
859     { \__template_assign_instance_aux:N \cs_gset_protected:Npn }
860     { \__template_assign_instance_aux:N \cs_set_protected:Npn }
861 }
862 \cs_new_protected:Npn \__template_assign_instance_aux:N #1
863 {
864   \tl_put_right:Ne \l__template_assignments_tl
865   {
866     \exp_not:N #1 \exp_not:V \l__template_var_tl
867     {
868       \__template_use_instance:nn
869       { \exp_not:V \l__template_keytype_arg_tl }
870       { \exp_not:V \l__template_value_tl }
871     }
872   }
873 }

```

(End of definition for __template_assign_instance: and __template_assign_instance_aux:N.)

__template_assign_variable: A general-purpose function for all of the other assignments. As long as the value is not coming from another variable, the stored value is simply transferred for output. We use V-type expansion for the \KeyValue case: for token lists this is essential, whilst for register-based variables, it does no harm and avoids needing a low-level test.

```

874 \cs_new_protected:Npn \__template_assign_variable:

```

```

875 {
876   \exp_args:Ne \__template_assign_variable:n
877   {
878     \__template_map_var_type:
879     -
880     \bool_if:NT \l__template_global_bool { g }
881     set:N
882   }
883 }

```

Notice we need a V-type variant for each (g)set operation here: these need to be provided by expl3.

```

884 \cs_new_protected:Npn \__template_assign_variable:n #1
885 {
886   \__template_if_key_value:VTF \l__template_value_tl
887   {
888     \__template_key_to_value:
889     \tl_put_right:Ne \l__template_assignments_tl
890     {
891       \exp_not:c { #1 V } \exp_not:V \l__template_var_tl
892       \exp_not:V \l__template_value_tl
893     }
894   }
895   {
896     \tl_put_right:Ne \l__template_assignments_tl
897     {
898       \exp_not:c { #1 n } \exp_not:V \l__template_var_tl
899       { \exp_not:V \l__template_value_tl }
900     }
901   }
902 }

```

(End of definition for __template_assign_variable: and __template_assign_variable:n.)

`__template_key_to_value:` The idea here is to recover the attribute value of another key. To do that, the marker is removed and a look up takes place. If this is successful, then the name of the variable of the attribute is returned. This assumes that the value will be used in context where it will be converted to a value, for example when setting a number. There is also a need to check in case the copied value happens to be global.

```

903 \cs_new_protected:Npn \__template_key_to_value:
904 { \exp_after:wN \__template_key_to_value_auxi:w \l__template_value_tl }
905 \cs_new_protected:Npn \__template_key_to_value_auxi:w \KeyValue #1
906 {
907   \tl_set:Ne \l__template_tmp_tl { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
908   \prop_get:NVNTF \l__template_vars_prop \l__template_tmp_tl
909   \l__template_value_tl
910   {
911     \exp_after:wN \__template_key_to_value_auxii:w \l__template_value_tl
912     \s__template_mark global \q__template_nil \s__template_stop
913   }
914   { \msg_error:nnV { template } { unknown-attribute } \l__template_tmp_tl }
915 }
916 \cs_new_protected:Npn \__template_key_to_value_auxii:w #1 global #2#3 \s__template_stop
917 {

```

```

918     \__template_quark_if_nil:NF #2
919     { \tl_set:Nn \l__template_value_tl {#2} }
920 }

```

(End of definition for `__template_key_to_value:`, `__template_key_to_value_auxi:w`, and `__template_key_to_value_auxii:w`.)

11.11 Using instances

```

\__template_use_instance:nn
  \__template_use_instance_aux:nNnnn
    \__template_use_instance_aux:nn

```

Using an instance is just a question of finding the appropriate function. If nothing is found, an error is raised. One complication is that if the first token of argument #2 is `\UseTemplate` then that is also valid. There is an error-test to make sure that the types agree, and if so the template is used directly.

```

921 \cs_new_protected:Npn \__template_use_instance:nn #1#2
922 {
923   \__template_if_use_template:nTF {#2}
924   { \__template_use_instance_aux:nNnnn {#1} #2 }
925   { \__template_use_instance_aux:nn {#1} {#2} }
926 }
927 \cs_new_protected:Npn \__template_use_instance_aux:nNnnn #1#2#3#4#5
928 {
929   \str_if_eq:nnTF {#1} {#3}
930   { \__template_use_template:nnn {#3} {#4} {#5} }
931   { \msg_error:nnnn { template } { type-mismatch } {#1} {#3} }
932 }
933 \cs_new_protected:Npn \__template_use_instance_aux:nn #1#2
934 {
935   \__template_if_instance_exist:nnTF {#1} {#2}
936   { \use:c { \c__template_instances_root_tl #1 / #2 } }
937   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
938 }

```

(End of definition for `__template_use_instance:nn`, `__template_use_instance_aux:nNnnn`, and `__template_use_instance_aux:nn`.)

11.12 Assignment manipulation

A few functions to transfer assignments about, as this is needed by `\AssignTemplateKeys`.

```

\__template_assignments_pop:

```

To actually use the assignments.

```

939 \cs_new:Npn \__template_assignments_pop: { \l__template_assignments_tl }

```

(End of definition for `__template_assignments_pop:.`)

```

\__template_assignments_push:n

```

Here, the assignments are stored for later use.

```

940 \cs_new_protected:Npn \__template_assignments_push:n #1
941 { \tl_set:Nn \l__template_assignments_tl {#1} }

```

(End of definition for `__template_assignments_push:n`.)

11.13 Showing templates and instances

`__template_show_code:nn` Showing the code for a template is just a translation of `\cs_show:c`.

```
942 \cs_new_protected:Npn \__template_show_code:nn #1#2
943 { \cs_show:c { \c__template_code_root_tl #1 / #2 } }
```

(End of definition for `__template_show_code:nn`.)

`__template_show_defaults:nn` A modified version of the property-list printing code, such that the output refers to templates and instances rather than to the underlying structures.

```
\__template_show_keytypes:nn
\__template_show_vars:nn
\__template_show:Nnnn
944 \cs_new_protected:Npn \__template_show_defaults:nn #1#2
945 {
946   \__template_if_keys_exist:nnT {#1} {#2}
947   {
948     \__template_recover_defaults:nn {#1} {#2}
949     \__template_show:Nnnn \l__template_values_prop
950     {#1} {#2} { default~values }
951   }
952 }
953 \cs_new_protected:Npn \__template_show_keytypes:nn #1#2
954 {
955   \__template_if_keys_exist:nnT {#1} {#2}
956   {
957     \__template_recover_keytypes:nn {#1} {#2}
958     \__template_show:Nnnn \l__template_keytypes_prop
959     {#1} {#2} { interface }
960   }
961 }
962 \cs_new_protected:Npn \__template_show_vars:nn #1#2
963 {
964   \__template_execute_if_code_exist:nnT {#1} {#2}
965   {
966     \__template_recover_vars:nn {#1} {#2}
967     \__template_show:Nnnn \l__template_vars_prop
968     {#1} {#2} { variable~mapping }
969   }
970 }
971 \cs_new_protected:Npn \__template_show:Nnnn #1#2#3#4
972 {
973   \msg_show:nneeee { template } { show~attribute }
974   { \tl_to_str:n {#2} }
975   { \tl_to_str:n {#3} }
976   { \tl_to_str:n {#4} }
977   { \prop_map_function:NN #1 \msg_show_item_unbraced:nn }
978 }
```

(End of definition for `__template_show_defaults:nn` and others.)

`__template_show_values:nn` Instance values are a little more complex, as is the template to consider.

```
979 \cs_new_protected:Npn \__template_show_values:nn #1#2
980 {
981   \__template_if_instance_exist:nnT {#1} {#2}
982   {
983     \__template_recover_values:nn {#1} {#2}
```

```

984     \msg_show:nneee { template } { show-values }
985     { \tl_to_str:n {#1} }
986     { \tl_to_str:n {#2} }
987     {
988         \prop_map_function:NN \l__template_values_prop
989         \msg_show_item_unbraced:nn
990     }
991 }
992 }

```

(End of definition for `__template_show_values:nn`.)

11.14 Messages

The text for error messages: short and long text for all of them.

```

993 \msg_new:nnnn { template } { argument-number-mismatch }
994 { Template~type~'~#1'~takes~#2~argument(s). }
995 {
996     Templates-of~type~'~#1'~require~#2~argument(s).\
997     You-have~tried~to~make~a~template~for~'~#1'~
998     with~#3~argument(s),~which~is~not~possible:~
999     the~number~of~arguments~must~agree.
1000 }
1001 \msg_new:nnnn { template } { bad-number-of-arguments }
1002 { Bad~number~of~arguments~for~template~type~'~#1'. }
1003 {
1004     A~template~may~accept~between~0~and~9~arguments.\
1005     You~asked~to~use~#2~arguments:~this~is~not~supported.
1006 }
1007 \msg_new:nnnn { template } { bad-variable }
1008 { Incorrect~variable~description~'~#1'. }
1009 {
1010     The~argument~'~#1'~is~not~of~the~form \
1011     ~'~<variable>' \
1012     ~or~ \
1013     ~'~global-<variable>' \
1014     It~must~be~given~in~one~of~these~formats~to~be~used~in~a~template.
1015 }
1016 \msg_new:nnnn { template } { choice-not-implemented }
1017 { The~choice~'~#1'~has~no~implementation. }
1018 {
1019     Each~choice~listed~in~the~interface~for~a~template~must~
1020     have~an~implementation.
1021 }
1022 \msg_new:nnnn { template } { choice-no-code }
1023 { The~choice~'~#1'~requires~implementation~details. }
1024 {
1025     When~creating~template~code~using~\DeclareTemplateCode,~
1026     each~choice~name~must~have~an~associated~implementation.\
1027     This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
1028 }
1029 \msg_new:nnnn { template } { choice-requires-code }
1030 { The~choice~'~#2'~for~key~'~#1'~requires~an~implementation. }
1031 {

```

```

1032     You-should-have-put:\\
1033     \\ \ #1::~choice-{\#2 = <code> ~} \\
1034     but~LaTeX~did~not~find~any~<code>.
1035 }
1036 \msg_new:nnnn { template } { duplicate-key-interface }
1037 { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }
1038 {
1039     Each-key~can~only~have~one~interface~declared~in~a~template.\\
1040     LaTeX~found~two~interfaces~for~'#1'.
1041 }
1042 \msg_new:nnnn { template } { keytype-requires-argument }
1043 { The-key~type~'#1'~requires~an~argument~\msg_line_context:. }
1044 {
1045     You-should-have-put:\\
1046     \\ \ <key-name>::~#1-{\<argument>} \\
1047     but~LaTeX~did~not~find~an~<argument>.
1048 }
1049 \msg_new:nnnn { template } { invalid-keytype }
1050 { The-key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1051 {
1052     Each-key~in~a~template~requires~a~key~type,~given~in~the~form:\\
1053     \\ \ <key>::~<key-type>\\
1054     LaTeX~could~not~find~a~<key-type>~in~your~input.
1055 }
1056 \msg_new:nnnn { template } { key-no-value }
1057 { The-key~'#1'~has~no~value~\msg_line_context:. }
1058 {
1059     When~creating~an~instance~of~a~template~
1060     every~key~listed~must~include~a~value:\\
1061     \\ \ <key>::~<value>
1062 }
1063 \msg_new:nnnn { template } { key-no-variable }
1064 { The-key~'#1'~requires~implementation~details~\msg_line_context:. }
1065 {
1066     When~creating~template~code~using~\DeclareTemplateCode,~
1067     each~key~name~must~have~an~associated~implementation.\\
1068     This~should~be~given~after~a~'= '~sign:~LaTeX~did~not~find~one.
1069 }
1070 \msg_new:nnnn { template } { key-not-implemented }
1071 { Key~'#1'~has~no~implementation~\msg_line_context:. }
1072 {
1073     The~definition~of~key~implementations~for~template~'#2'~
1074     of~template~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
1075     The~key~was~declared~in~the~interface~definition,~
1076     and~so~an~implementation~is~required.
1077 }
1078 \msg_new:nnnn { template } { missing-keytype }
1079 { The-key~'#1'~is~missing~a~key~type~\msg_line_context:. }
1080 {
1081     Key~interface~definitions~should~be~of~the~form\\
1082     \\ \ #1::~<key-type>\\
1083     but~LaTeX~could~not~find~a~<key-type>.
1084 }
1085 \msg_new:nnnn { template } { no-template-code }

```

```

1086 {
1087     The~template~'#2'~of~type~'#1'~is~unknown~
1088     or~has~no~implementation.
1089 }
1090 {
1091     There~is~no~code~available~for~the~template~name~given.\\
1092     This~should~be~given~using~\DeclareTemplateCode.
1093 }
1094 \msg_new:nnnn { template } { type-already-defined }
1095 { Template~type~'#1'~already~defined. }
1096 {
1097     You~have~used~\NewTemplateType~
1098     with~a~template~type~that~has~already~been~defined.
1099 }
1100 \msg_new:nnnn { template } { type-mismatch }
1101 { Template~types~'#1'~and~'#2'~do~not~agree. }
1102 {
1103     You~are~trying~to~use~a~template~directly~with~\UseInstance
1104     (or~a~similar~function),~but~the~template~types~do~not~match.
1105 }
1106 \msg_new:nnnn { template } { unknown-attribute }
1107 { The~template~attribute~'#1'~is~unknown. }
1108 {
1109     There~is~a~definition~in~the~current~template~reading\\
1110     \\ \token_to_str:N \KeyValue {~#1~} \\
1111     but~there~is~no~key~called~'#1'.
1112 }
1113 \msg_new:nnnn { template } { unknown-choice }
1114 { The~choice~'#2'~was~not~declared~for~key~'#1'. }
1115 {
1116     The~key~'#1'~takes~a~fixed~list~of~choices~
1117     and~this~list~does~not~include~'#2'.
1118 }
1119 \msg_new:nnnn { template } { unknown-default-choice }
1120 { The~default~choice~'#2'~was~not~declared~for~key~'#1'. }
1121 {
1122     The~key~'#1'~takes~a~fixed~list~of~choices~
1123     and~this~list~does~not~include~'#2'.
1124 }
1125 \msg_new:nnnn { template } { unknown-expansion }
1126 { The~expansion~type~'#1'~is~unknown. }
1127 {
1128     Key~values~can~only~be~expanded~using~one~of~the~pre~defined~methods:~
1129     n,~o,~V,~v,~e,~N~or~c.
1130 }
1131 \msg_new:nnnn { template } { unknown-instance }
1132 { The~instance~'#2'~of~type~'#1'~is~unknown. }
1133 {
1134     You~have~asked~to~use~an~instance~'#2',~
1135     but~this~has~not~been~created.
1136 }
1137 \msg_new:nnnn { template } { unknown-key }
1138 { Unknown~template~key~'#1'. }
1139 {

```

```

1140     The~key~'#1'~was~not~declared~in~the~interface~
1141     for~the~current~template.
1142 }
1143 \msg_new:nnnn { template } { unknown-keytype }
1144 { The~key~type~'#1'~is~unknown. }
1145 {
1146     Valid~key~types~are:\\
1147     --boolean;\\
1148     --choice;\\
1149     --commalist;\\
1150     --function;\\
1151     --instance;\\
1152     --integer;\\
1153     --length;\\
1154     --muskip;\\
1155     --real;\\
1156     --skip;\\
1157     --tokenlist.
1158 }
1159 \msg_new:nnnn { template } { unknown-type }
1160 { The~template~type~'#1'~is~unknown. }
1161 {
1162     A~template~type~needs~to~be~defined~with~\NewTemplateType
1163     prior~to~using~it.
1164 }
1165 \msg_new:nnnn { template } { unknown-template }
1166 { The~template~'#2'~of~type~'#1'~is~unknown. }
1167 {
1168     No~interface~has~been~declared~for~a~template~
1169     '#2'~of~template~type~'#1'.
1170 }

```

Information messages only have text: more text should not be needed.

```

1171 \msg_new:nnn { template } { declare-instance }
1172 { Declaring~instance~'#1'~of~type~'#2'~\msg_line_context:. }
1173 \msg_new:nnn { template } { declare-template-code }
1174 { Declaring~code~for~template~'#2'~of~template~type~'#1'~\msg_line_context:. }
1175 \msg_new:nnn { template } { declare-template-interface }
1176 {
1177     Declaring~interface~for~template~'#2'~of~template~type~'#1'~
1178     \msg_line_context:.
1179 }
1180 \msg_new:nnn { template } { declare-type }
1181 { Declaring~template~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1182 \msg_new:nnn { template } { show-attribute }
1183 {
1184     The~template~'#2'~of~type~'#1'~has~
1185     \tl_if_empty:nTF {#4} { no~#3. } { #3 : #4 }
1186 }
1187 \msg_new:nnn { template } { show-values }
1188 {
1189     The~instance~'#2'~of~type~'#1'~has~
1190     \tl_if_empty:nTF {#3} { no~values. } { values: #3 }
1191 }

```


Also add template to the LaTeX messages.

```
1192 \prop_gput:Nnn \g_msg_module_type_prop { template } { LaTeX }
```

11.15 User functions

All simple translations.

```
\NewTemplateType
\DeclareTemplateInterface 1193 \cs_new_protected:Npn \NewTemplateType #1#2
\DeclareTemplateCode      1194 { \__template_define_type:nn {#1} {#2} }
\DeclareTemplateCopy     1195 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
\EditTemplateDefaults   1196 { \__template_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
\UseTemplate             1197 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
\DeclareInstance        1198 { \__template_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
\DeclareInstanceCopy    1199 \cs_new_protected:Npn \DeclareTemplateCopy #1#2#3
\EditInstance           1200 { \__template_template_set_eq:nnn {#1} {#2} {#3} }
\UseInstance            1201 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3
                        1202 { \__template_edit_defaults:nnn {#1} {#2} {#3} }
                        1203 \cs_new_protected:Npn \UseTemplate #1#2#3
                        1204 { \__template_use_template:nnn {#1} {#2} {#3} }
                        1205 \cs_new_protected:Npn \DeclareInstance #1#2#3#4
                        1206 { \__template_declare_instance:nnnn {#1} {#3} {#2} {#4} }
                        1207 \cs_new_protected:Npn \DeclareInstanceCopy #1#2#3
                        1208 { \__template_instance_set_eq:nnn {#1} {#2} {#3} }
                        1209 \cs_new_protected:Npn \EditInstance #1#2#3
                        1210 { \__template_edit_instance:nnn {#1} {#2} {#3} }
                        1211 \cs_new_protected:Npn \UseInstance #1#2
                        1212 { \__template_use_instance:nn {#1} {#2} }
```

(End of definition for \NewTemplateType and others. These functions are documented on page 3.)

The show functions are again just translation.

```
\ShowTemplateCode
\ShowTemplateDefaults 1213 \cs_new_protected:Npn \ShowTemplateCode #1#2
\ShowTemplateInterface 1214 { \__template_show_code:nn {#1} {#2} }
\ShowTemplateVariables 1215 \cs_new_protected:Npn \ShowTemplateDefaults #1#2
\ShowInstanceValues   1216 { \__template_show_defaults:nn {#1} {#2} }
                        1217 \cs_new_protected:Npn \ShowTemplateInterface #1#2
                        1218 { \__template_show_keytypes:nn {#1} {#2} }
                        1219 \cs_new_protected:Npn \ShowTemplateVariables #1#2
                        1220 { \__template_show_vars:nn {#1} {#2} }
                        1221 \cs_new_protected:Npn \ShowInstanceValues #1#2
                        1222 { \__template_show_values:nn {#1} {#2} }
```

(End of definition for \ShowTemplateCode and others. These functions are documented on page 10.)

More direct translation.

```
\IfInstanceExistsT 1223 \cs_new:Npn \IfInstanceExistsTF #1#2
\IfInstanceExistsF 1224 { \__template_if_instance_exist:nnTF {#1} {#2} }
\IfInstanceExistsTF 1225 \cs_new:Npn \IfInstanceExistsT #1#2
                        1226 { \__template_if_instance_exist:nnT {#1} {#2} }
                        1227 \cs_new:Npn \IfInstanceExistsF #1#2
                        1228 { \__template_if_instance_exist:nnF {#1} {#2} }
```

(End of definition for \IfInstanceExistsT, \IfInstanceExistsF, and \IfInstanceExistsTF. These functions are documented on page 8.)

\KeyValue Simply dump the argument when executed: this should not happen.

```
1229 \cs_new_protected:Npn \KeyValue #1 {#1}
```

(End of definition for \KeyValue. This function is documented on page 4.)

\AssignTemplateKeys A short call to use a token register by proxy.

```
1230 \cs_new_protected:Npn \AssignTemplateKeys { \__template_assignments_pop: }
```

(End of definition for \AssignTemplateKeys. This function is documented on page 5.)

\SetKnownTemplateKeys A friendly wrapper, with some speed up for the common case of the third argument being empty.
\SetTemplateKeys

```
1231 \cs_new_protected:Npn \SetKnownTemplateKeys #1#2#3
1232 {
1233   \tl_if_empty:oTF {#3}
1234   {
1235     \tl_set_eq:NN \UnusedTemplateKeys \c_empty_tl
1236   }
1237   {
1238     \keys_set:known:no { template / #1 / #2 } {#3} \UnusedTemplateKeys
1239   }
1240 }
1241 \cs_new_protected:Npn \SetTemplateKeys #1#2#3
1242 {
1243   \tl_if_empty:oF {#3}
1244   {
1245     \keys_set:no { template / #1 / #2 } {#3}
1246   }
1247 }
1248 \tl_new:N \UnusedTemplateKeys
```

(End of definition for \SetKnownTemplateKeys and \SetTemplateKeys. These functions are documented on page 6.)

```
1249 <latexrelease>\IncludeInRelease{0000/00/00}{lttemplates}%
1250 <latexrelease>          {Prototype-document-commands}%
1251 <latexrelease>
1252 <latexrelease>\EndModuleRelease
```

```
1253 \ExplSyntaxOff
1254 </2ekernel | latexrelease>
```

We need to stop DocStrip treating @@ in a special way at this point.

```
1255 <@@=)
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
\\	996, 1004, 1010, 1011, 1012, 1013, 1026, 1032, 1033, 1039, 1045, 1046, 1052, 1053, 1060, 1067, 1074, 1081, 1082, 1091, 1109, 1110, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156
_	1033, 1046, 1053, 1061, 1082, 1110
A	
\AssignTemplateKeys ...	<u>1230</u> , 390, 395, 6
B	
bool commands:	
\bool_if:NTF	858, 880, 223, 229, 689, 792, 841
\bool_new:N	21, 22
\bool_set_false:N	278, 687, 768
\bool_set_true:N ...	249, 291, 623, 777
\l_tmpa_bool	5
C	
\c	386
\caption	2
clist commands:	
\clist_if_empty:NTF	542, 571
\clist_if_in:NnTF	267, 578
\clist_map_inline:Nn	544
\clist_new:N	33
\clist_remove_all:Nn	580
\clist_set:Nn	532
\l_tmpa_clist	5
cs commands:	
\cs_generate_from_arg_count:NNnn	405, 467, 849
\cs_generate_variant:Nn	65, 350, 510, 660, 662, 734, 765, 838
\cs_gset:Npn	842
\cs_gset_eq:NN	673
\cs_gset_protected:Npn	859, 407
\cs_if_exist:NTF	55, 61, 74, 87, 403, 461, 496, 671, 694, 710
\cs_if_exist_use:NTF	651, 759
\cs_new:Npn	939, 1223, 1225, 1227, 351, 352, 462, 516
\cs_new_eq:NN ..	357, 358, 359, 374, 661
\cs_new_protected:Npe	276
\cs_new_protected:Npn	856, 862, 874, 884, 903, 905, 916, 921, 927, 933,
\cs_set:Npn	940, 942, 944, 953, 962, 971, 979, 1193, 1195, 1197, 1199, 1201, 1203, 1205, 1207, 1209, 1211, 1213, 1215, 1217, 1219, 1221, 1229, 1230, 1231, 1241, 43, 53, 59, 66, 72, 97, 105, 121, 129, 137, 147, 163, 173, 183, 189, 204, 220, 242, 254, 271, 297, 321, 353, 355, 360, 362, 364, 366, 368, 370, 372, 375, 401, 409, 420, 422, 436, 440, 454, 511, 530, 548, 569, 586, 593, 601, 606, 615, 621, 626, 641, 648, 658, 663, 676, 685, 704, 717, 729, 735, 741, 746, 766, 774, 779, 790, 796, 816, 833, 839, 845
\cs_set:Npn	329, 843
\cs_set_eq:NN	712
\cs_set_protected:Npe	696
\cs_set_protected:Npn	860, 325
\cs_show:N	943, 36
D	
debug commands:	
\debug_resume:	103, 119, 127, 135
\debug_suspend:	99, 107, 123, 131
\DeclareInstance	<u>1193</u> , 8
\DeclareInstanceCopy	<u>1193</u> , 8
\DeclareTemplateCode	1025, 1066, 1092, <u>1193</u> , 5
\DeclareTemplateCopy	<u>1193</u> , 7
\DeclareTemplateInterface	<u>1193</u> , 4
dim commands:	
\dim_eval:n	365
\dim_new:N	34
\l_tmpa_dim	5
E	
\EditInstance	<u>1193</u> , 9
\EditTemplateDefaults	<u>1193</u> , 9
\EndModuleRelease	1252
exp commands:	
\exp_after:wN	904, 911, 286, 771
\exp_args:Ne	876
\exp_not:N	851, 866, 891, 898, 278, 279, 280, 281, 291, 297, 300, 302, 303, 307, 309, 312, 317, 468, 469, 484, 486, 502, 632, 641, 642, 698, 700, 803, 811

<ul style="list-style-type: none"> \exp_not:n 852, 853, 866, 869, 870, 891, 892, 898, 899, 283, 471, 596, 699, 804, 805, 812, 850 \ExplSyntaxOff 1253 \ExplSyntaxOn 6 	<ul style="list-style-type: none"> \msg_new:nnn 1171, 1173, 1175, 1180, 1182, 1187 \msg_new:nnnn 993, 1001, 1007, 1016, 1022, 1029, 1036, 1042, 1049, 1056, 1063, 1070, 1078, 1085, 1094, 1100, 1106, 1113, 1119, 1125, 1131, 1137, 1143, 1159, 1165 \msg_show:nnnnn 984 \msg_show:nnnnnn 973 \msg_show_item_unbraced:nn 977, 989
F	
fp commands:	
<ul style="list-style-type: none"> \fp_eval:n 369 \l_tmpa_fp 5 	
I	
<ul style="list-style-type: none"> \IfInstanceExistsF <u>1223</u>, 8 \IfInstanceExistsT <u>1223</u>, 8 \IfInstanceExistsTF <u>1223</u>, 8 \IncludeInRelease 1249 	
int commands:	
<ul style="list-style-type: none"> \int_compare:nNnTF 46 \int_compare:nTF 192 \int_eval:n 363 \int_new:N 35 \int_set:Nn 191 \l_tmpa_int 5 \item 6 	
K	
kernel internal commands:	
<ul style="list-style-type: none"> __kernel_quark_new_conditional:Nn 42 	
keys commands:	
<ul style="list-style-type: none"> \keys_define:nn 513, 534, 595 \keys_set:nn 1245 \keys_set_known:nnN 1238 	
keyval commands:	
<ul style="list-style-type: none"> \keyval_parse:NNn 213, 618 \keyval_parse:nnn 414, 535 \KeyValue 905, 1110, <u>1229</u>, 80, 33 	
M	
\message 3	
msg commands:	
<ul style="list-style-type: none"> \msg_error:nn 268 \msg_error:nnn 914, 63, 70, 186, 234, 248, 292, 314, 421, 434, 451, 507, 545, 624, 637, 653, 762 \msg_error:nnnn 931, 937, 57, 76, 200, 563, 591, 603, 715, 727, 827 \msg_error:nnnnn 49, 418 \msg_info:nnnn 110, 194, 404, 672, 695, 711 \msg_line_context: 1037, 1043, 1050, 1057, 1064, 1071, 1079, 1172, 1174, 1178, 1181 \g_msg_module_type_prop 1192 	<ul style="list-style-type: none"> \msg_new:nnn 1171, 1173, 1175, 1180, 1182, 1187 \msg_new:nnnn 993, 1001, 1007, 1016, 1022, 1029, 1036, 1042, 1049, 1056, 1063, 1070, 1078, 1085, 1094, 1100, 1106, 1113, 1119, 1125, 1131, 1137, 1143, 1159, 1165 \msg_show:nnnnn 984 \msg_show:nnnnnn 973 \msg_show_item_unbraced:nn 977, 989
	muskip commands:
	<ul style="list-style-type: none"> \muskip_eval:n 367 \muskip_new:N 36 \l_tmpa_muskip 5
	N
	<ul style="list-style-type: none"> \NewModuleRelease 7 \NewTemplateType 1097, 1162, <u>1193</u>, 3
	P
	prg commands:
	<ul style="list-style-type: none"> \prg_generate_conditional_ variant:Nnn 84 \prg_new_conditional:Npnn 78, 85, 91 \prg_return_false: 82, 89, 95 \prg_return_true: 81, 88, 94
	prop commands:
	<ul style="list-style-type: none"> \prop_clear:N 145, 155, 171, 181, 210, 211, 413 \prop_clear_new:N 124 \prop_gclear:N 111 \prop_gclear_new:N 100, 132 \prop_get:NnN 45, 558, 561, 588, 722, 743, 824 \prop_get:NnNTF 908, 426, 539, 634, 748, 750, 835 \prop_gput:Nnn 1192, 196 \prop_gset_eq:NN 101, 114, 133 \prop_if_exist:NTF 108, 139, 149, 165, 175 \prop_if_in:NnTF 68, 185, 552, 556 \prop_map_function:NN 977, 988 \prop_map_inline:Nn 417 \prop_new:N 18, 29, 31, 32, 113 \prop_put:Nnn 262, 354, 356, 361, 475, 489, 504, 533, 599, 691 \prop_remove:Nn 432 \prop_set_eq:NN 125, 142, 152, 168, 178
	Q
	quark commands:
	<ul style="list-style-type: none"> \q_nil 80, 93 \quark_new:N 41 \q_stop 80, 93, 632, 642

quark internal commands:
 \q__template_nil 912, 41, 13

R

regex commands:
 \regex_match:nnTF 386

S

scan commands:
 \scan_new:N 39, 40

scan internal commands:
 \s__template_mark 912, 39, 12
 \s__template_stop
 912, 916, 40, 287, 298, 309,
 330, 344, 352, 438, 441, 771, 774, 12

\section 2

seq commands:
 \seq_clear:N 161, 212
 \seq_const_from_clist:Nn 16
 \seq_gclear_new:N 116
 \seq_gset_eq:NN 117
 \seq_if_exist:NTF 156
 \seq_if_in:NnTF 231
 \seq_map_break: 250, 341
 \seq_map_function:NN .. 227, 347, 738
 \seq_new:N 30
 \seq_put_right:Nn 264
 \seq_set_eq:NN 158

\SetKnownTemplateKeys 1231, 6
 \SetTemplateKeys 1231, 6
 \ShowInstanceValues 1213, 10
 \ShowTemplateCode 1213, 10
 \ShowTemplateDefaults 1213, 10
 \ShowTemplateInterface 1213, 10
 \ShowTemplateVariables 1213, 10

skip commands:
 \skip_eval:n 371
 \skip_new:N 37
 \l_tmpa_skip 5

str commands:
 \str_case:nn 518
 \str_case:nnTF 456
 \str_if_eq:nnTF 929, 80, 93,
 244, 265, 470, 485, 501, 573, 753, 755
 \str_if_in:nnTF 384
 \str_new:N 28
 \str_set:Nn 646

T

template internal commands:
 __template_assign_boolean: 790, 790
 __template_assign_boolean_aux:n
 790, 793, 794, 796
 __template_assign_choice: . 816, 816
 __template_assign_choice_-
 aux:nTF 816, 818, 821, 833, 838
 __template_assign_function: 839, 839
 __template_assign_function_-
 aux:N 839, 842, 843, 845
 __template_assign_instance: 856, 856
 __template_assign_instance_-
 aux:N 856, 859, 860, 862
 __template_assign_variable: ...
 874, 874, 760
 __template_assign_variable:n ...
 874, 876, 884
 __template_assignments_pop: ...
 939, 939, 1230
 __template_assignments_push:n ..
 940, 940, 698
 \l__template_assignments_tl
 864, 889, 896, 939, 941,
 19, 699, 737, 801, 809, 836, 847, 11
 \c__template_code_root_tl
 9, 943, 55,
 403, 406, 671, 673, 674, 700, 787, 11
 __template_convert_to_assignments:
 693, 735, 735, 786
 __template_convert_to_assignments_-
 aux:n 735, 739, 741
 __template_convert_to_assignments_-
 aux:nn 735, 744, 746, 765
 __template_declare_instance:nnnn
 1206, 676, 676
 __template_declare_instance_-
 aux:nnnn 676, 682, 685, 732
 __template_declare_template_-
 code:nnnn .. 375, 387, 389, 394, 401
 __template_declare_template_-
 code:nnnn 1198, 375, 375
 __template_declare_template_-
 keys:nnnn 1196, 204, 204
 __template_declare_type:nn
 183, 187, 189
 \l__template_default_tl 20, 11
 \c__template_defaults_root_tl ...
 10, 100, 101, 140, 143, 11
 __template_define_type:nn
 1194, 183, 183
 __template_edit_defaults:nnn ...
 1202, 606, 606
 __template_edit_instance:nnn ...
 1210, 717, 717
 __template_edit_instance_-
 aux:nnnn 724, 729, 734
 __template_edit_instance_-
 aux:nnnnn 717

```

\l__template_error_bool . 21, 223,
    229, 249, 278, 291, 623, 687, 689, 12
\__template_execute_if_arg_-
agree:nnTF . . . . . 43, 43, 208, 379
\__template_execute_if_code_-
exist:nnTF . . . . . 964, 53, 53, 678, 781
\__template_execute_if_keys_-
exist:nnTF . . . . . 72
\__template_execute_if_keytype_-
exist:nTF . . . . . 59, 59, 65, 225
\__template_execute_if_type_-
exist:nTF . . . . . 66, 66, 206, 377
\__template_find_global: 756, 766, 766
\__template_find_global_aux:w . . .
. . . . . 766, 771, 774
\l__template_global_bool . . . . .
. . . . . 858, 880, 22, 768, 777, 792, 841, 12
\__template_if_instance_exist:nn 85
\__template_if_instance_exist:nnTF
935, 981, 1224, 1226, 1228, 85, 706, 719
\__template_if_key_value:n . . . . . 78, 84
\__template_if_key_value:nTF . . .
. . . . . 886, 78, 798
\__template_if_keys_exist:nnTF . .
. . . . . 946, 955, 72, 381, 608
\__template_if_use_template:n . . . . . 91
\__template_if_use_template:nTF .
. . . . . 923, 91
\__template_implement_choice_-
elt:n . . . . . 536, 569, 601
\__template_implement_choice_-
elt:nnn . . . . . 537, 569, 569
\__template_implement_choice_-
elt_aux:n . . . . . 569, 575, 583, 586
\__template_implement_choice_-
elt_aux:nnn . . . . . 569, 574, 581, 593
\__template_implement_choices:nn
. . . . . 458, 530, 530
\__template_implement_choices_-
default: . . . . . 530, 541, 548
\__template_instance_set_eq:nnn .
. . . . . 1208, 704, 704
\c__template_instances_root_tl . .
. . . . . 11, 936, 87, 694, 696, 710, 712, 713, 11
\l__template_key_name_tl . . . . .
. . . . . 23, 232, 235, 262, 264, 285,
300, 307, 312, 354, 356, 361, 424,
427, 432, 476, 490, 505, 514, 533,
534, 539, 551, 555, 558, 561, 564,
565, 588, 591, 596, 598, 604, 634,
637, 645, 758, 819, 822, 824, 828, 27
\c__template_key_order_root_tl . .
. . . . . 13, 116, 117, 156, 159, 11
\l__template_key_order_seq . . . . . 30,
118, 158, 161, 212, 231, 264, 738, 12
\__template_key_to_value: . . . . .
. . . . . 888, 903, 903, 800
\__template_key_to_value_auxi:w .
. . . . . 903, 904, 905
\__template_key_to_value_auxii:w
. . . . . 903, 911, 916
\l__template_keytype_arg_tl . . . .
. . . . . 852, 869, 25, 246,
259, 260, 267, 324, 338, 471, 532, 12
\l__template_keytype_tl . . . . . 24, 225,
244, 258, 265, 274, 323, 334, 428,
430, 456, 518, 659, 753, 755, 759, 12
\c__template_keytypes_arg_seq . . .
. . . . . 16, 227, 347, 11
\l__template_keytypes_prop . . . . . 958,
29, 115, 152, 155, 211, 262, 417,
426, 432, 558, 588, 634, 743, 824, 12
\c__template_keytypes_root_tl . . . .
. . . . . 74, 108, 111, 113, 114, 150, 153, 11
\__template_map_var_type: . . . . .
. . . . . 878, 497, 500, 516, 516
\__template_parse_keys_elt:n . . .
. . . . . 214, 220, 220, 273, 19
\__template_parse_keys_elt:nn . . .
. . . . . 214, 271, 271
\__template_parse_keys_elt_aux: .
. . . . . 220, 237, 254
\__template_parse_keys_elt_aux:n
. . . . . 220, 228, 242
\__template_parse_values:nnn . . .
. . . . . 611, 615, 615, 688, 785
\__template_parse_values_elt:n . .
. . . . . 619, 621, 621
\__template_parse_values_elt:nn .
. . . . . 619, 626, 626
\__template_parse_values_elt_-
aux:n . . . . . 626, 636, 648
\__template_parse_values_elt_-
aux:w . . . . . 626, 630, 641
\__template_parse_values_exp:N . .
. . . . . 626, 661, 662
\__template_parse_values_exp:n . .
. . . . . 626, 658, 660, 661
\__template_parse_vars_elt:n . . .
. . . . . 415, 420, 420
\__template_parse_vars_elt:nnn . .
. . . . . 415, 422, 422
\__template_parse_vars_elt_-
aux:nn . . . . . 422, 431, 436
\__template_parse_vars_elt_-
aux:nnn . . . . . 422, 444, 448, 454, 510

```

__template_parse_vars_elt_-	__template_store_value_commalist:n
aux:nw 422, 438, 440 360, 374
__template_parse_vars_elt_-	__template_store_value_function:n
key:nn 422, 463, 480, 498, 511 355, 358
__template_quark_if_nil:N 42	__template_store_value_instance:n
__template_quark_if_nil:NTF 918 355, 359
__template_quark_if_nil:nTF 42	__template_store_value_integer:n
__template_quark_if_nil_p:n 42 360, 362
__template_recover_defaults:nn .	__template_store_value_length:n
948, 137, 137, 411, 610, 665, 680, 783 360, 364
__template_recover_keytypes:nn .	__template_store_value_muskip:n
. 957, 137, 147, 412, 617, 667 360, 366
__template_recover_values:nn . . .	__template_store_value_real:n . .
. 983, 137, 163, 708, 721 360, 368
__template_recover_vars:nn	__template_store_value_skip:n . .
. 966, 137, 173, 669, 681, 731, 784 360, 370
\c__template_restrict_root_tl 11	__template_store_value_tokenlist:n
__template_show:Nnnn 360, 372, 374
. 944, 949, 958, 967, 971	__template_store_values:nn
__template_show_code:nn 97, 121, 692, 709
. 942, 942, 1214	__template_store_vars:nn
__template_show_defaults:nn 97, 129, 416, 670
. 944, 944, 1216	__template_template_set_eq:nnn .
__template_show_keytypes:nn 1200, 663, 663
. 944, 953, 1218	\l__template_tmp_clist
__template_show_values:nn 33, 532, 542, 544, 571, 578, 580, 12
. 979, 979, 1222	\l__template_tmp_dim 34, 12
__template_show_vars:nn	\l__template_tmp_int
. 944, 962, 1220 35, 191, 192, 195, 197, 201, 12
__template_split_keytype:n	\l__template_tmp_muskip 36, 12
. 222, 276, 276	\l__template_tmp_skip 37, 12
__template_split_keytype_arg:n .	\l__template_tmp_tl 907,
. 317, 321,	908, 914, 38, 45, 46, 50, 256, 263,
321, 350, 430, 560, 590, 650, 752, 826	279, 280, 281, 287, 540, 550, 551,
__template_split_keytype_arg_-	552, 554, 555, 556, 559, 560, 562,
aux:n 321, 325, 348, 351	589, 590, 597, 599, 635, 650, 723,
__template_split_keytype_arg_-	725, 743, 744, 825, 826, 835, 836, 12
aux:w 321, 329, 344, 352	\g__template_type_prop
__template_split_keytype_aux:w 18, 45, 68, 185, 196, 11
. 276, 286, 297, 309	__template_use_instance:nn
__template_store_defaults:nn 868, 921, 921, 1212
. 97, 97, 215, 612, 666	__template_use_instance_aux:nn .
__template_store_key_implementation:nnn 921, 925, 933
. 383, 409, 409	__template_use_instance_-
__template_store_keytypes:nn	aux:nNnnn 921, 924, 927
. 97, 105, 216, 668	__template_use_template:nnn
__template_store_value:n 930, 1204, 779, 779
. 355, 355, 357, 358, 359	\l__template_value_exp_str
__template_store_value_aux:Nn 28, 646, 651, 653, 12
360, 360, 363, 365, 367, 369, 371, 373	\l__template_value_tl 853, 870,
__template_store_value_boolean:n	886, 892, 899, 904, 909, 911, 919,
. 353, 353	26, 748, 798, 805, 811, 819, 829, 12
__template_store_value_choice:n	\l__template_values_prop
. 355, 357 949, 988, 31, 102,

126, 142, 145, 168, 171, 210, 354, 356, 361, 539, 561, 691, 722, 748, 12	\tl_put_right:Nn 864, 889, 896, 300, 307, 801, 809, 836, 847
\c_template_values_root_tl	\tl_replace_all:Nnn 280
. 14, 124, 125, 166, 169, 11	\tl_set:Nn
\l_template_var_tl 907, 919, 941, 256, 279, 323, 334, 338, 424, 550, 554, 597, 645, 758, 776
. 866, 891, 898, 27, 750, 769, 771, 776, 804, 812, 850, 12	\tl_set_eq:NN 1235
\l_template_vars_prop . . 908, 967, 32, 134, 178, 181, 413, 475, 489, 504, 533, 552, 556, 599, 750, 835, 12	\tl_to_str:n 907, 974, 975, 976, 985, 986, 303, 425, 631
\c_template_vars_root_tl	\tl_trim_spaces:n
. 15, 132, 133, 176, 179, 11 907, 302, 323, 335, 425, 449, 631
tl commands:	\l_tmpa_tl 5
\c_empty_tl 1235	token commands:
\c_space_tl . . . 551, 555, 598, 819, 822	\token_to_str:N
\tl_clear:N 285, 324, 737	1110, 280, 281, 298, 305, 308, 315, 642
\tl_const:Nn . . 9, 10, 11, 12, 13, 14, 15	
\tl_head:w 80, 93	U
\tl_if_blank:nTF . . . 332, 336, 443, 446	\UnusedTemplateKeys . . 1235, 1238, 1248, 6
\tl_if_empty:nTF 246, 259, 312	use commands:
\tl_if_empty:nTF 1185, 1190, 1233, 1243	\use:N 936, 274, 497, 659, 787
\tl_if_in:nnTF 281, 305, 327, 769	\use:n 295, 339, 373, 628, 639
\tl_if_single:nTF 494	\use_i:nn 5
\tl_new:N	\use_none:n 654
. . . 19, 20, 23, 24, 25, 26, 27, 38, 1248	\UseInstance 1103, 1193, 486, 9
	\UseTemplate 1193, 93, 9